# Input-queued switches: Scheduling algorithms for a crossbar switch

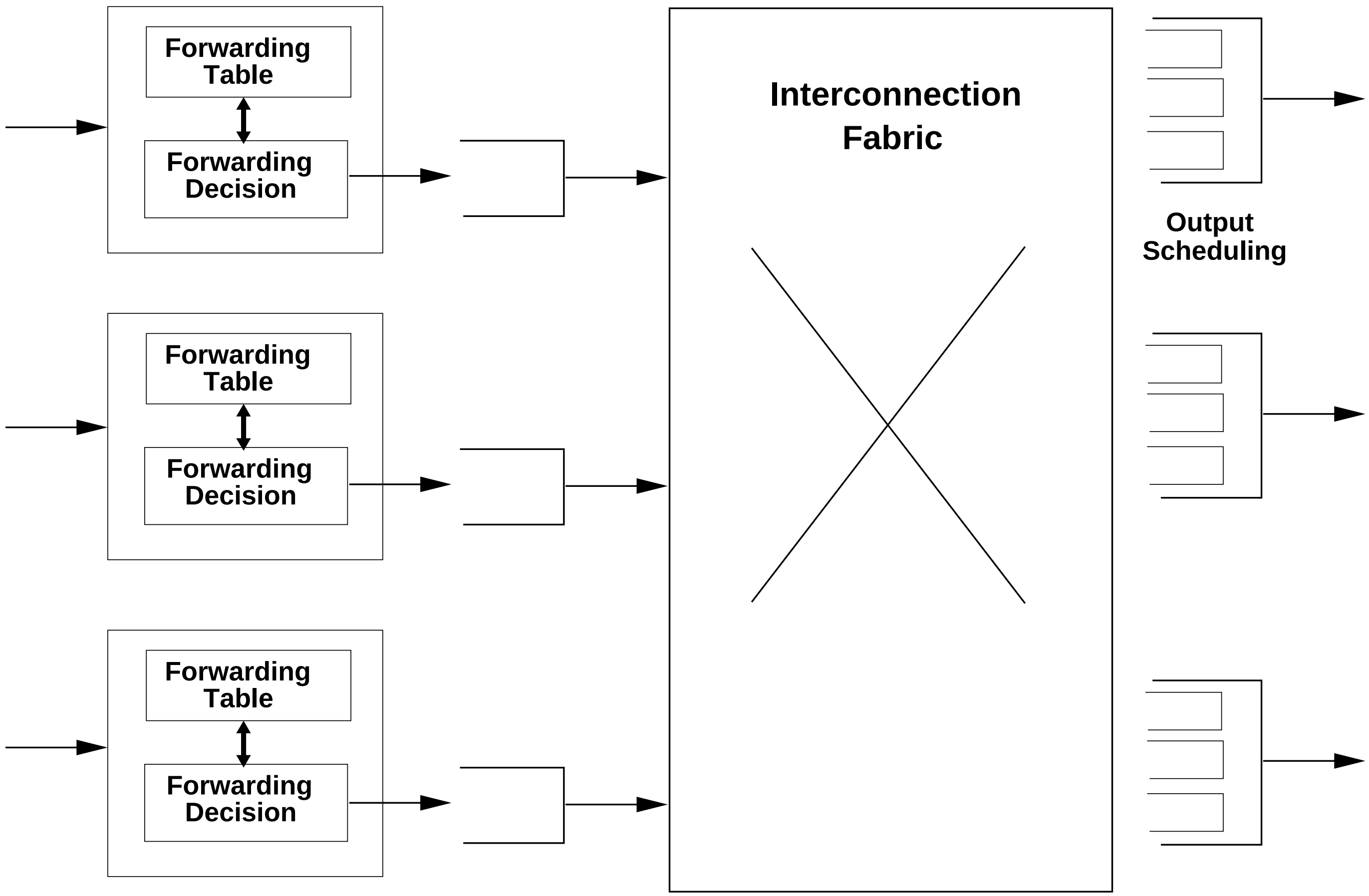
# Overview

- Today's lecture
  - the input-buffered switch architecture
  - the head-of-line blocking phenomenon
  - algorithms for 100% throughput

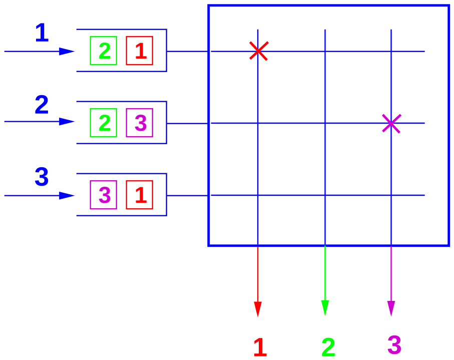
    (maximum weight matching algorithm, randomized versions)
- Subsequent lectures
  - more details on each of the above

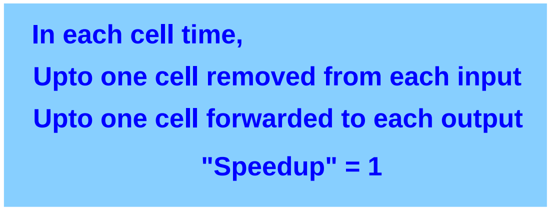
# A Detailed Look at Switching

- Packets arrive on line cards. The decision to route is made. Switching is done. Output scheduling follows. Packets are sent out.



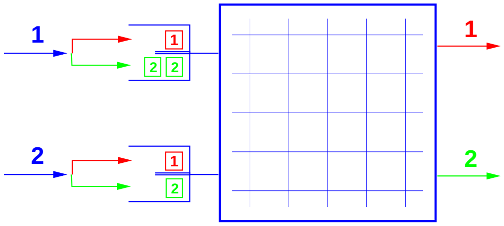
# Input-queued Switches



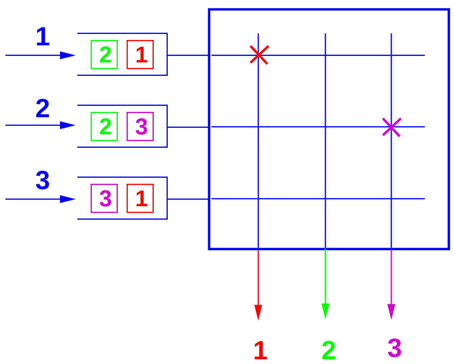

- Major problem: Head-of-line blocking limits throughput to 58%

- Overcoming HoL blocking: Use virtual output-queues




  - with this architectural change, we can get 100% throughput
  - but, first let's understand HoL blocking

# Head-of-line Blocking

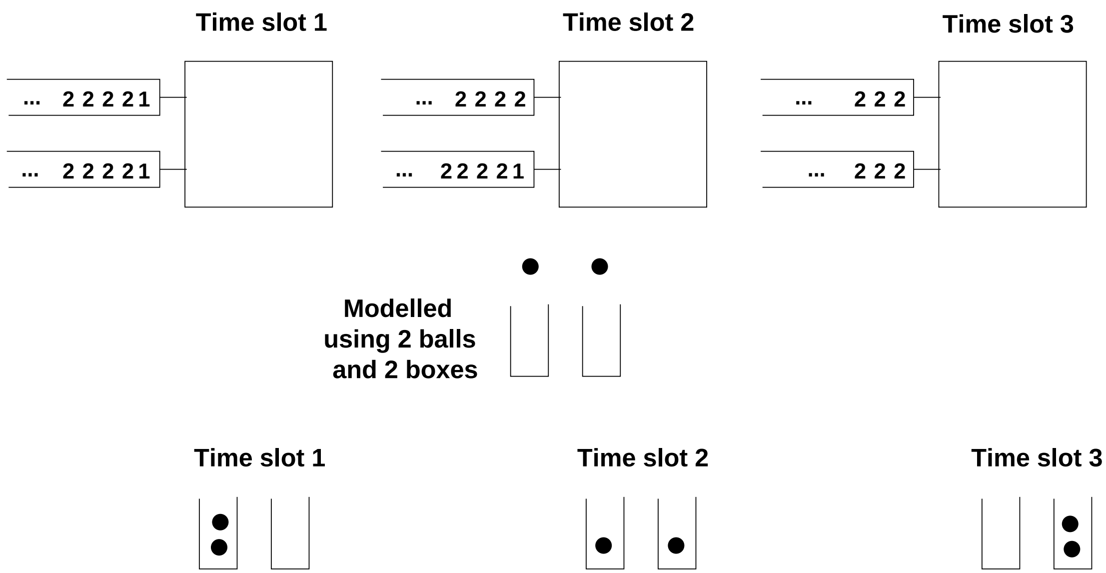



In each cell time,
Upto one cell removed from each input
Upto one cell forwarded to each output
"Speedup" = 1

- The setting: Consider an $N \times N$ input-queued switch
  - time is slotted, so that at most one packet can arrive (depart) per time slot
  - packets arrive at each input with probability $p$, independently across inputs/time
  - the destination of a packet is equally likely to be one of the outputs and independent across all packets
  - the "load matrix" $\{\lambda_{ij}\}$ equals $\{\frac{p}{N}\}$ for every $i$ and $j$
- The scheduling policy: At each time an output chooses one HoL packet u.a.r.
  - Question: What is the highest value of $p$ so that back-logs don't grow without bound?

# Head-of-line Blocking

- This is easy to understand using a 2×2 switch.




- Saturation analysis
  - an infinite number of packets are placed in both buffers initially
  - the numbers on the packets indicate the output they want to go to
  - the numbers are chosen independently and uniformly from $\{1, ..., N\}$
  - this ball-bin model can be used to determine the maximum throughput, $p$

- The ball-bin model
  - the bin on the left corresponds to output 1, that on the right to output 2
  - imagine there are two balls, each one corresponding to one of the HoL packets
  - at time 0 drop each ball independently into one of the bins u.a.r.
  - in each successive time slot do the following...

    **1.** remove at most one ball from each non-empty bin

    **2.** drop each ball-in-hand into one of the bins independently and u.a.r.
  - note that this process is a Markov chain
- The equivalence
  - you are sitting either at an output and recording whether a packet departed or not
  - throughput from an output $= P($a packet departs from it in equilibrium$)$

    $= P($the corresponding bin is non-empty in equilibrium$)$
- Throughput
  - from switch $= 1 \times P($both balls in same bin$) + 2 \times P($both balls in different bins$)$

    $= 1\ (2\ .\ \frac{1}{2}.\frac{1}{2}) + 2\ (2\ .\ \frac{1}{2}.\frac{1}{2}) = 1.5$
  - from output i: $1.5/2 = .75$ (by symmetry)

# Larger i/q switches

- We can determine the throughput of HoL blocked $N \times N$ i/q switches using Markov chains for *any* $N$

- But the problem is: state space explosion !

| Switch size, $N$ | # of states | Throughput |
|---|---|---|
| 1 | 1 | 1.00 |
| 2 | 2 | 0.75 |
| 3 | 3 | 0.6825 |
| 4 | 5 | 0.6552 |
| 5 | 7 | 0.64 |
| 6 | 11 | 0.6301 |

- the number of states grows like the partition function

- we can use a simple queueing-theoretic trick

# Throughput of HoL Blocked Switches

- Saturation analysis: use balls/boxes model with $N$ balls and $N$ boxes
  - focus on the first box (i.e. output 1)
  - let $X_t$ be number of balls box 1 at time $t =$ number of HoL packets for output 1
  - let $D_t^N$ be the number of balls removed from *all* boxes at end of time t

    (note that $D_t^N$ equals switch throughput at time $t$)

- Let
  - $A_{t+1}$ be number of balls dropping into box 1 at time $t+1$
  - $X_t$ satisfies the recursion $X_{t+1} = X_t + A_{t+1} - 1_{\{X_t>0\}}$, $\qquad (*)$

    where $P(A_{t+1} = k|D_t^N) = \binom{D_t^N}{k}\left(\frac{1}{N}\right)^k\left(1-\frac{1}{N}\right)^{D_t^N-k}$

- A useful approximation
  - $E(D_t^N)$ is the average switch throughput, the quantity we're interested in
  - let $E(D_t^N) = \rho N$, where $\rho$ is the average per output throughput
  - when $N$ is large enough, it is possible to show $P(D_t^N = \rho N) \approx 1$
  - so $A_t$ has a Poisson distribution: $P(A_t = k) \approx e^{-\rho}\frac{\rho^k}{k!}$

- Therefore $X_{t+1} = X_t + A_{t+1} - 1_{\{X_t > 0\}}$, where
  - $A_{t+1}$ is independent of $X_t$
  - $\{A_t\}$ is IID, Poisson($\rho$); therefore $E(A_t) = \rho$ and $E(A^2) = \rho + \rho^2$
  - Question: What is $\rho$?
- Take expectations at equation $(*)$
  - and hit steady-state to drop the $t$ subscript
  - we get: $EX = EX + EA - P(X > 0)$ or that $EA = P(X > 0)$
- We want $E(X)$
  - so, square equation $(*)$ and take expectations to get

$$E(X^2) = E(X^2) + E(A^2) + P(X > 0) + 2E(AX) - 2\,E(X\,1_{\{X>0\}}) - 2E(A\,1_{\{X>0\}})$$

  - but, on the RHS, $A$ and $X$ are independent
  - using this to simplify we get $EX = \frac{E(A^2)\ +\ P(X>0)\ -\ 2\,EA\,P(X>0)}{2\,(1-EA)} = \frac{E(A^2)\ +\ EA\ -\ 2\,(EA)^2}{2\,(1-EA)}$
  - since $EA = \rho$, $E(A^2) = \rho + \rho^2$, we get

$$EX = \frac{2\rho - \rho^2}{2\,(1 - \rho)}$$

- We can find $\rho$ if we know what $EX$ is ...

  - but, $EX = 1$ because there are exactly $N$ balls and $N$ boxes !

    that is, the average number of balls in box 1 equals 1 at all times

  - solving the quadratic

$$1 = \frac{2\rho - \rho^2}{2\,(1-\rho)}$$

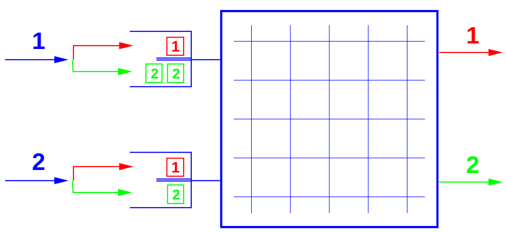
  - gives $\rho = 2 - \sqrt{2} \approx 58.6\%$

  $\longrightarrow$ this is a famous result in switching, due to Karol et. al. (1987)

- Thus, to eliminate HoL blocking, we need to change the FIFO organization of the input buffers

  - at input $i$, use a separate queue for the packets destined for output $j$

  - this queue is denoted VOQ$_{ij}$



# Notation

- Consider an $N \times N$ input-queued switch with VOQs.
  - let $A_{ij}(n)$ indicate the packet arrivals at input $i$ for output $j$
  - that is, $A_{ij}(n) = 1$ if a packet arrived at input $i$ for output $j$ in time slot $n$
  - let $\{A_{ij}(n)\}$ be IID across $i$, $j$ and $n$
  - let $\lambda_{ij} = E(A_{ij}(n))$ be the arrival rate; note
  - given the line rate $L$, $\{A_{ij}(n)\}$ is said to be *admissible* if
    - $\sum_j \lambda_{ij} < L$ for every $i$: no input is oversubscribed
    - $\sum_i \lambda_{ij} < L$ for every $j$: no output is oversubscribed
  - let $q_{ij}(n))$ be the queue-size (number of back-logged packets) in VOQ$_{ij}$ at time $n$
- Schedule at time $n$: $S(n)$
  - a schedule or matching at time $n$ is a decision to connect input-output pairs so that no input (output) is connected to more than one output (input)

  $\longrightarrow$ this a direct consequence of using a crossbar interconnection fabric
  - let $S_{ij}(n)$ indicate whether input $i$ and output $j$ are connected at time $n$
  - thus, $S(n) = \{S_{ij}(n)\}$ is a permutation matrix

- Scheduling algorithm
  - is a rule that determines schedules $S(n)$ for every $n$
  - it can do this either by knowing the traffic matrix: $\Lambda = \{\lambda_{ij}\}$
  - or by merely knowing $Q(n) = \{q_{ij}(n)\}$
  - most switches are designed to work for the second case (since $\Lambda$ is usually unknown)
- Goals for designing good scheduling algorithms

  **1.** 100% throughput: ensure that $sup_{n,i,j} E\left[q_{ij}(n)\right] < \infty$ so long as input is admissible

  - thus, what comes in will (eventually) go out if no input/output is oversubscribed

  **2.** minimize back-logs, delays: minimize $sup_{n,i,j} E\left[q_{ij}(n)\right]$
- Thus, switch scheduling is
  - designing input-output matching algorithms
  - either by knowing $\Lambda$ or $Q(n)$
  - so as to achieve high throughputs and low delays/back-logs
  - a notational convenience: we'll normalize $L = 1$ so that

$$\sum_i \lambda_{ij} \leq 1, \quad \sum_j \lambda_{ij} \leq 1$$

for all admissible traffic

# Suppose $\Lambda$ is known

- Some facts about $\Lambda$
  - first note that it is doubly sub-stochastic, and not necessarily uniform ($\lambda_{ij} \neq c$)
    (i.e. each row and column sum is less than 1, all entries are non-negative)
  - Fact 0: a doubly sub-stochastic matrix is majorized by a suitable doubly stochastic matrix
    (there exists a $\Lambda' = \{\lambda'_{ij}\}$ such that $\lambda_{ij} < \lambda'_{ij}$ and $\sum_i \lambda'_{ij} = 1 = \sum_j \lambda'_{ij}$)
  - Fact 1: the set of all doubly stochastic matrices is convex
  - Fact 2: any convex, compact set in $R^n$ has extreme points
    (Facts 0 and 1 are trivially true, Fact 2 is deeper.)
  - Theorem (Birkhoff-von Neumann): Permutation matrices are the extreme points of the set of doubly stochastic matrices.

- Use this as follows
  - given $\Lambda$, we find a suitable doubly stochastic $\Lambda'$ to dominate it
  - then we decompose $\Lambda' = \sum_{k=1}^{K} \alpha_k P^k$,
    where $\sum_{k=1}^{K} \alpha_k = 1$ and $\alpha_k > 0$, and $P^k$ are permutation matrices

- Scheduling algorithm
  - let $C$ be a $K$-sided coin with $P(C = k) = \alpha_k$
  - at time $n$, flip $C$ and let $S(n) = P^k$ if $C = k$
  - note that

$$P(S_{ij}(n) = 1) = \sum_{k=1}^{K} P(S_{ij}(n) = 1 | C = k) P(C = k) = \sum_{k=1}^{K} P_{ij}^k \alpha_k = \lambda'_{ij}$$

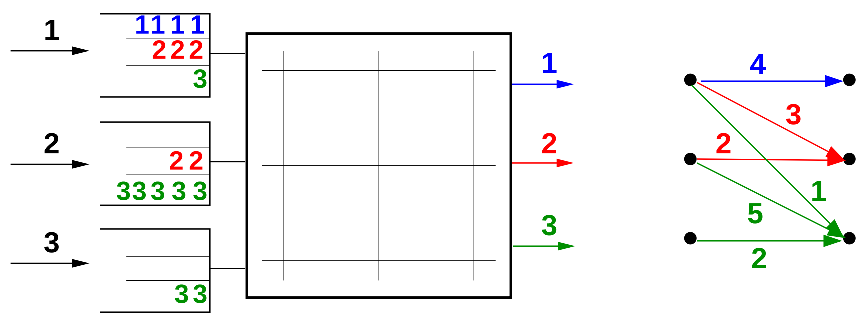
- Proof that this algorithm gives 100% throughput
  - since $q_{ij}(n) = [q_{ij}(n-1) + A_{ij}(n) - S_{ij}(n)]^+$
  - we see that $q_{ij}(n)$ is a simple birth-death Markov chain
  - with birth rate $= P(A_{ij}(n) = 1) = E\left[A_{ij}(n)\right] = \lambda_{ij}$
  - and death rate $= P(S_{ij}(n) = 1) = \lambda'_{ij} > \lambda_{ij}$

  $\rightarrow$ therefore, the chain is ergodic and $E\left[q_{ij}(n)\right] < \infty$ QED

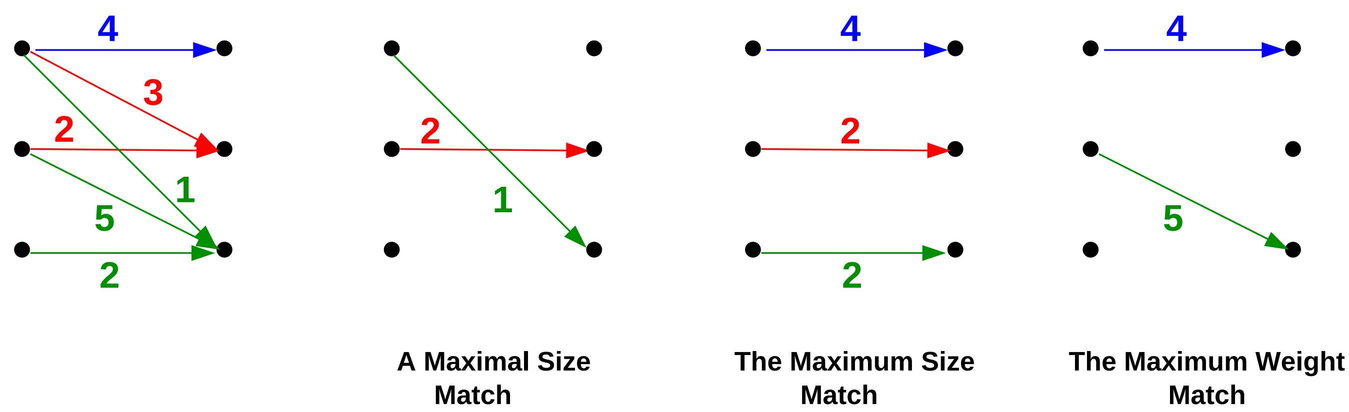
- In summary
  - this simple algorithm gives 100% throughput
  - note that it may not minimze back-logs/delays (in fact, it is quite poor)
  - it is easy to implement the algorithm
  - we will lose this feature (implementation gets harder) when we don't know $\Lambda$

# Suppose $\Lambda$ is unknown

- Then our scheduling algorithm will use $Q(n)$
  - the switch scheduling problem becomes a bipartite graph matching problem
  - i.e. consider a $N \times N$ bipartite graph
  - the edge, $e_{ij}(n)$, is present between i/p $i$ and o/p $j$ at time $n$ iff $q_{ij}(n) > 0$
  - the weight, $w_{ij}(n)$, of $e_{ij}(n)$ is some increasing function of $q_{ij}(n)$; e.g. $w_{ij}(n) = q_{ij}(n)$




- Designing scheduling algorithms becomes finding matchings in this bipartite graph
  - so that we get 100% throughput for all admissible $\Lambda$ (which is unknown)
  - and, the average back-log or delay is minimized
  - Question: What matchings should we find?




- Trade-offs
  - the maximal size matching is easiest to implement (in fact, this is done in practice)

    $\longrightarrow$ but, it doesn't give 100% throughput, unless one uses a higher speedup (next class)
  - the maximum size matching is harder to implement (esp because of augmenting paths)

    $\longrightarrow$ surprisingly, it doesn't give 100% throughput either!
  - the maximum weight matching is also hard to implement (because of augmenting paths)

    $\longrightarrow$ but, it does give 100% throughput!

- Question:
  - how does one *prove* that the max weight matching algorithm gives 100% throughput?
  - need to use Lyapunov functions (aka potential functions)

# Discussion of scheduling algorithms

- We have seen that max wt and max size matchings are not suitable for high-speed implementations. Let's understand why.
  - basically at line rates of 10 gbps (OC 192 lines), packets arrive roughly 40-50 ns apart
  - this means scheduling decisions need to be made at this speed
  - now, finding maximum weight matchings takes roughly $O(N^3)$ iterations
  - this means for a 30 port switch, the worst-case number of iterations is about 27,000
- But, time complexity is not the biggest problem
  - we may be able to overcome this with hardware optimizations
  - the bigger problem is that it is hard to pipeline the max wt matching routine
  - that is, if the decisions made in each iteration about which edges to include in the final matching were binding, then the iterations can be executed serially
  - but, the augmenting path routine used in max wt matching algorithms prevents this i.e. we will have to backtrack
  - so the entire pipeline is held up while the matching for time slot $n$ is decided

- This is easy to understand using an example of the opposite kind
  - let's consider the "greedy maximum weight matching"; called iLQF
  - given a weighted bipartite graph, in each iteration choose the heaviest edge from the residual graph (break ties at random); you'll be done in $N$ iterations
  - $\rightarrow$ the key point here is that future iterations do not disconnect currently chosen edges
  - $\rightarrow$ this is pipelineable
  - the matching found by iLQF is maximal; and its weight $\geq 0.5 \times$ max wt matching
  - annoyingly (or interestingly), iLQF does not give 100% throughput!
- Let's formalize this class of "greedy, maximal" matchings which are implmentation-friendly
  - essentially, most switch schedulers follow the "request-grant-accept" (RGA) routine
  - this requires an input (output) to rank all the outputs (inputs) from 1 to $N$
  - e.g. input $i$ ranks output $j$ higher than output $k$ if $q_{ij}(n) > q_{ik}(n)$
  - given these ranking lists, run the "stable marriage" routine
  - this is essentially the RGA routine