CS112: Data Structures

Lecture 13

Schedule

- Monday, August 8:
  - Work on project 4
- Wednesday, August 10:
  - Review
- Monday August 15:
  - Students present Projects 4 (attendance required)
- Wednesday, August 17:
  - Final exam

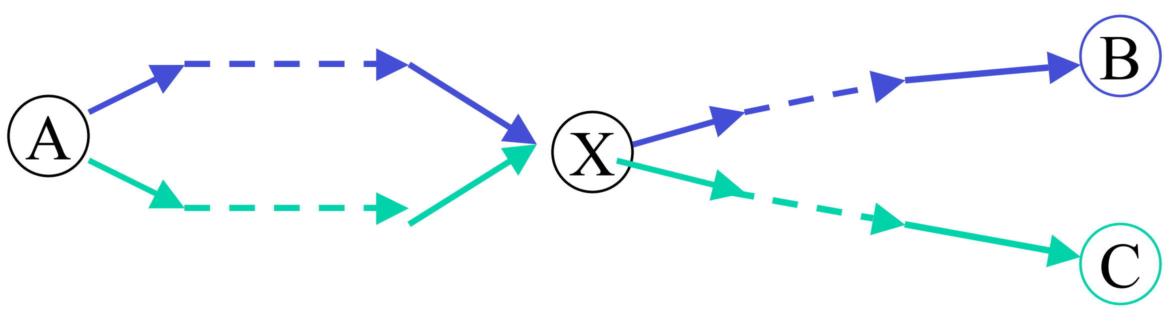
Review: Shortest Path

Dijkstra's algorithm:

to find shortest path from A to B:

- **Build a tree of shortest paths from A**
  - **Is set of shortest paths really a tree?**

    Suppose not, then must have two shortest paths converge and then diverge



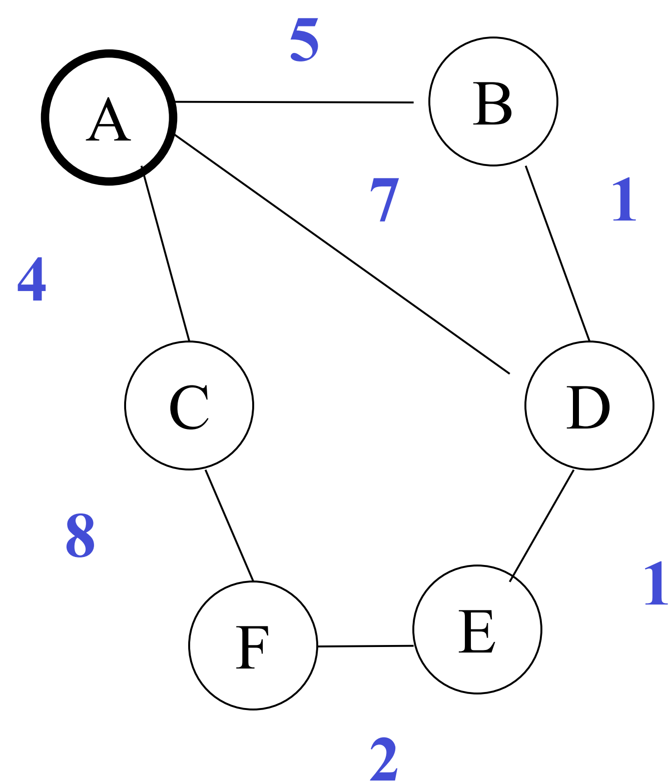
Dijkstra's algorithm

Grow a tree of shortest paths from start

- **grow it one edge / vertex at a time**
- **But which?**
  - **Vertex has to be one edge from tree**
  - **Of edges for a vertex, has to be edge that gives shortest path to start**
  - **Of vertices one edge from tree, choose the one with the shortest 'shortest path via tree'**

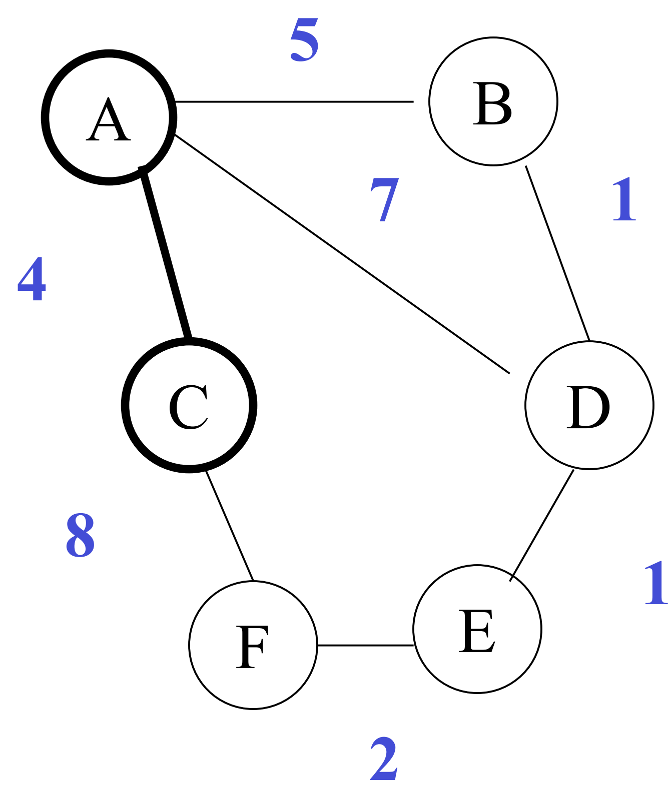
Example

Node	Status	Lin K	Distance
A	Tree	--	0
B	Fringe	A	5
C	Fringe	A	4
D	Fringe	A	7
E			
F			



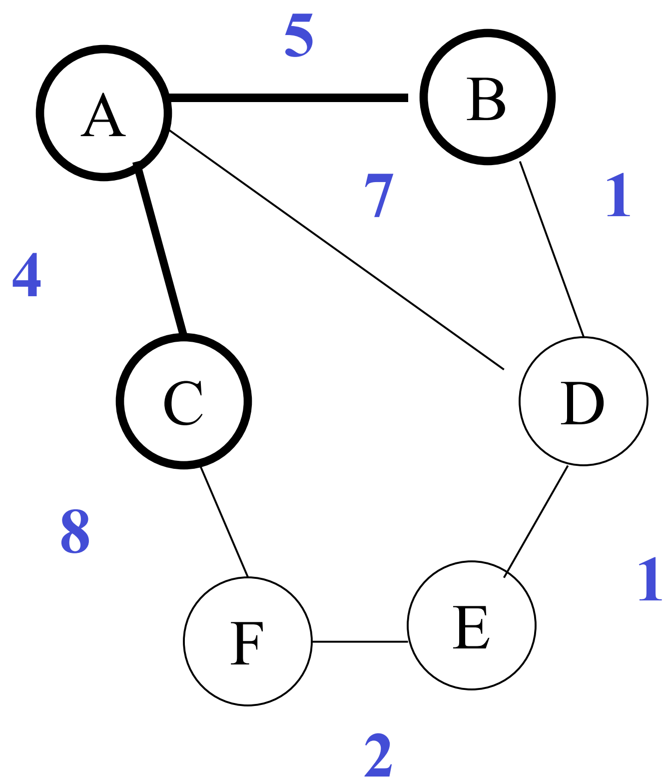
Example

Node	Status	Lin K	Distance
A	Tree	--	0
B	Fringe	A	5
C	Tree	A	4
D	Fringe	A	7
E			
F	Fringe	C	12



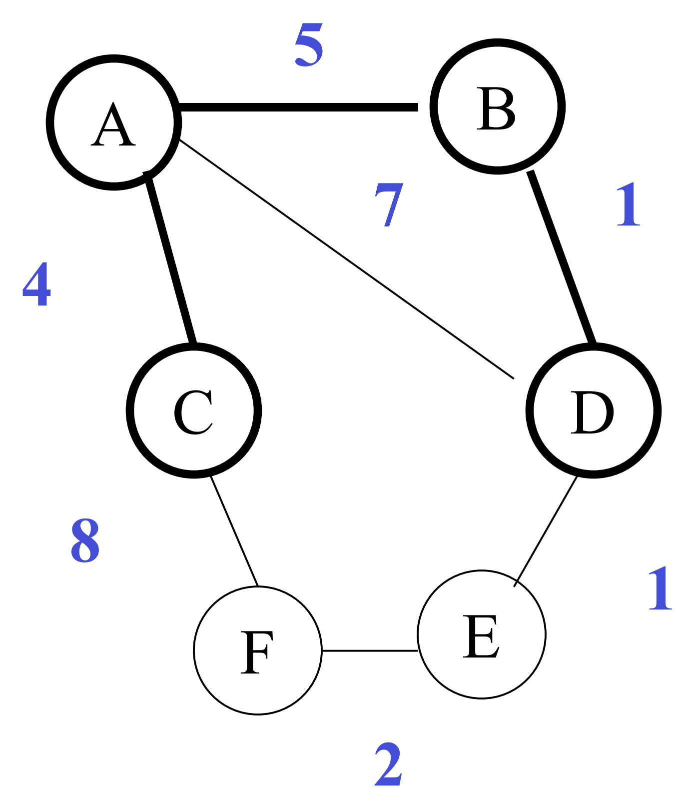
Example

Node	Status	Lin K	Distance
A	Tree	--	0
B	Tree	A	5
C	Tree	A	4
D	Fringe	B	6
E			
F	Fringe	C	12



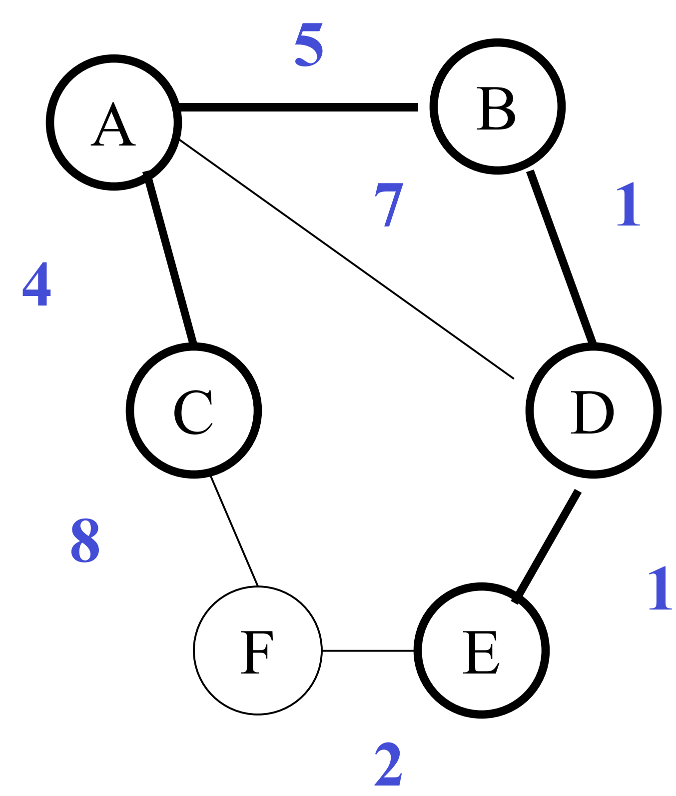
Example

Node	Status	Lin K	Distance
A	Tree	--	0
B	Tree	A	5
C	Tree	A	4
D	Tree	B	6
E	Fringe	D	7
F	Fringe	C	12



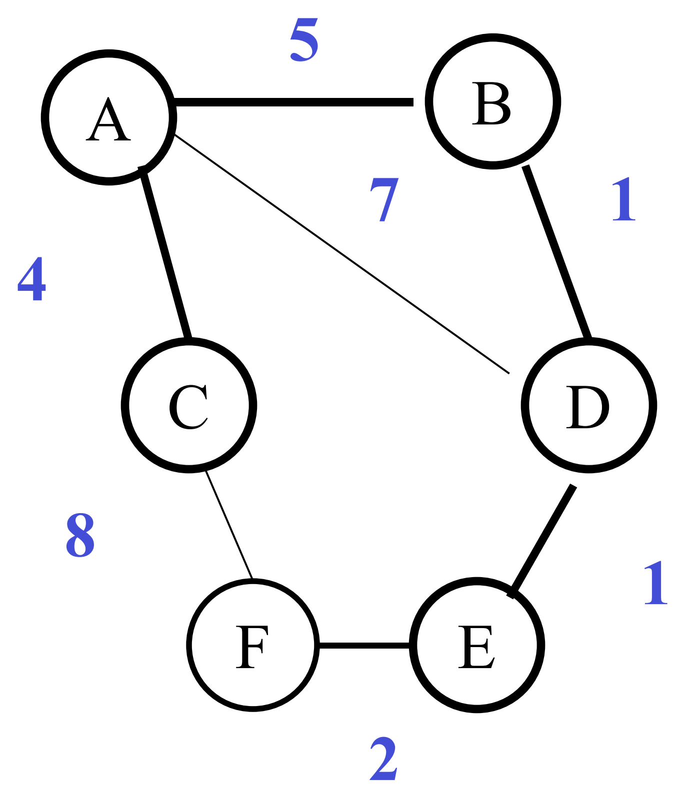
Example

Node	Status	Lin K	Distance
A	Tree	--	0
B	Tree	A	5
C	Tree	A	4
D	Tree	B	6
E	Tree	D	7
F	Fringe	E	9



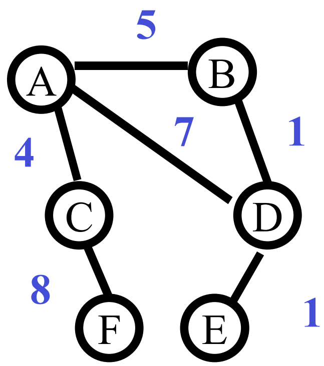
Example

Node	Status	Link	Distance
A	Tree	--	0
B	Tree	A	5
C	Tree	A	4
D	Tree	B	6
E	Tree	D	7
F	Tree	E	9

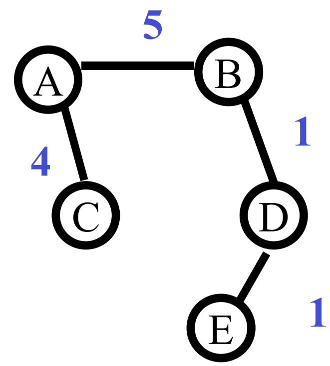


Minimum Spanning Tree

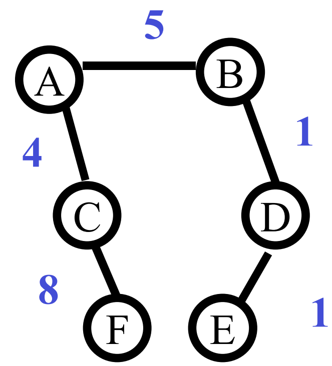
- **Spanning Tree: a subgraph with**
  - **All the nodes**
  - **Some of the edges**
  - **A tree, I.E., one path between any pair of nodes**
- **Minimum spanning tree**
  - **A spanning tree**
  - **With minimum total edge weight**



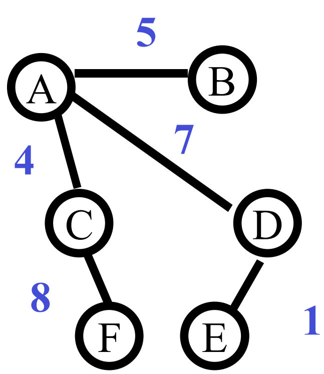
Not a tree (has a cycle)



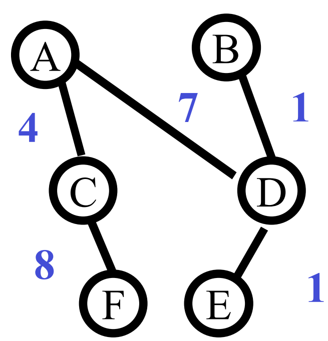
Not spanning (leaves out node F)



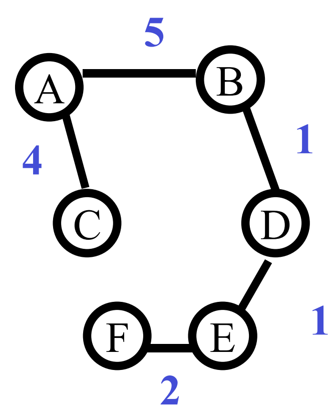
Not minimal



Not minimal



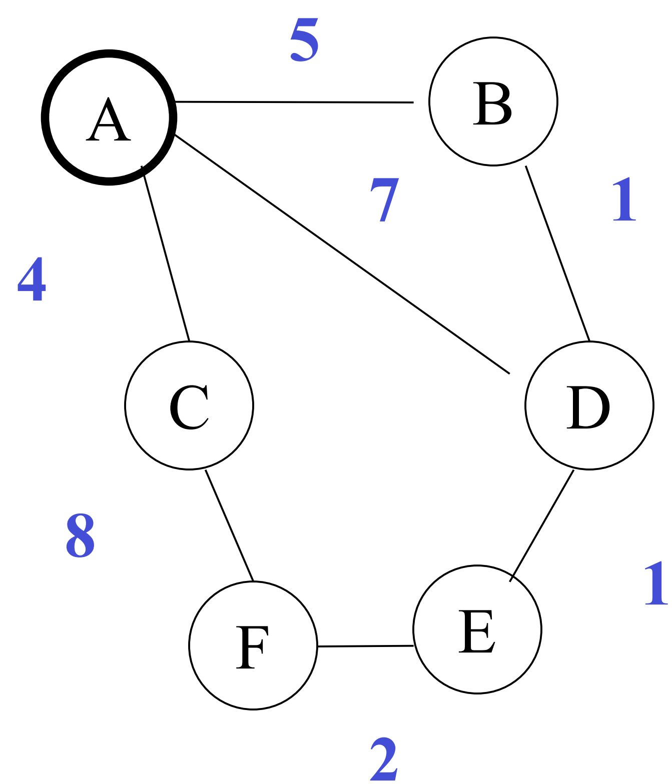
Not minimal



Minimal Spanning Tree

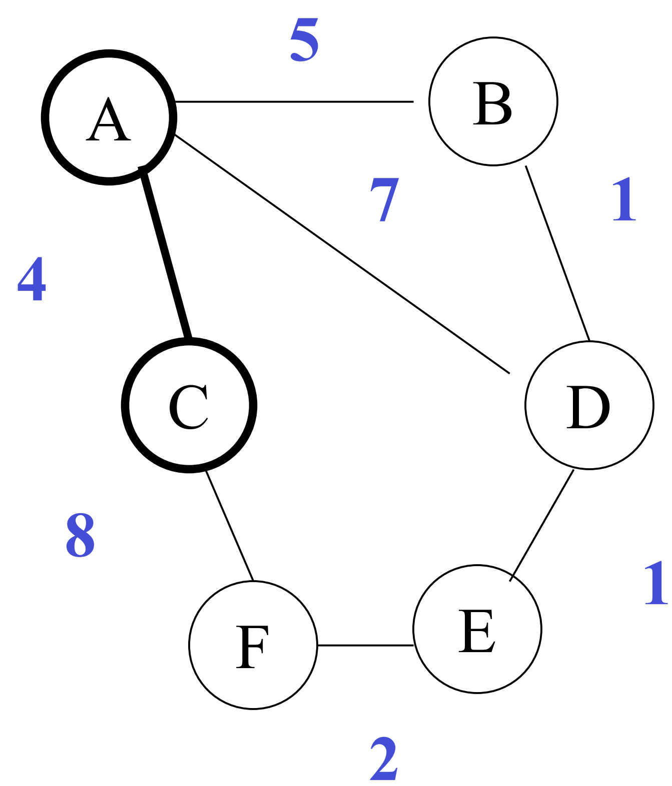
Example

Node	Status	Link	Weight
A	Tree	--	0
B	Fringe	A	5
C	Fringe	A	4
D	Fringe	A	7
E			
F			



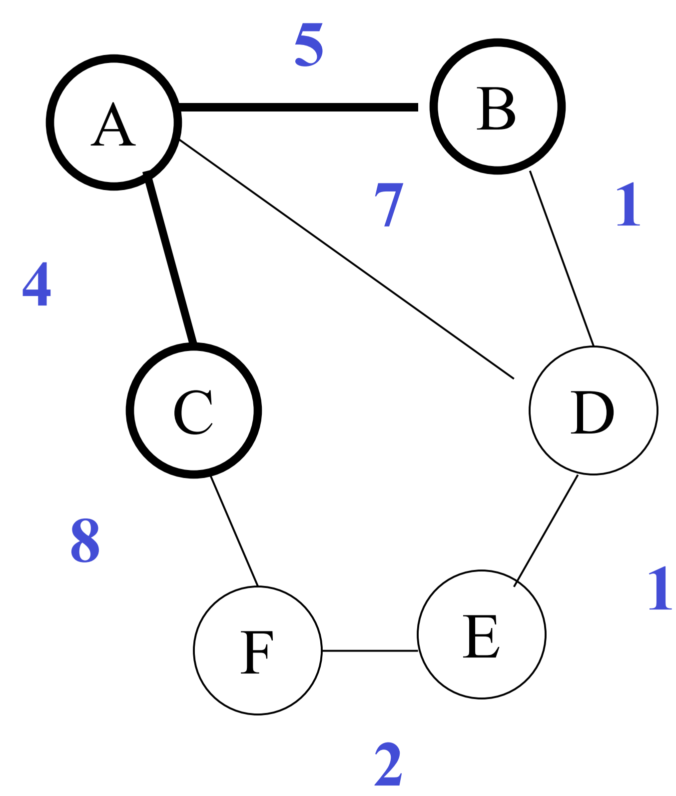
Example

Node	Status	Link	Weight
A	Tree	--	0
B	Fringe	A	5
C	Tree	A	4
D	Fringe	A	7
E			
F	Fringe	C	8



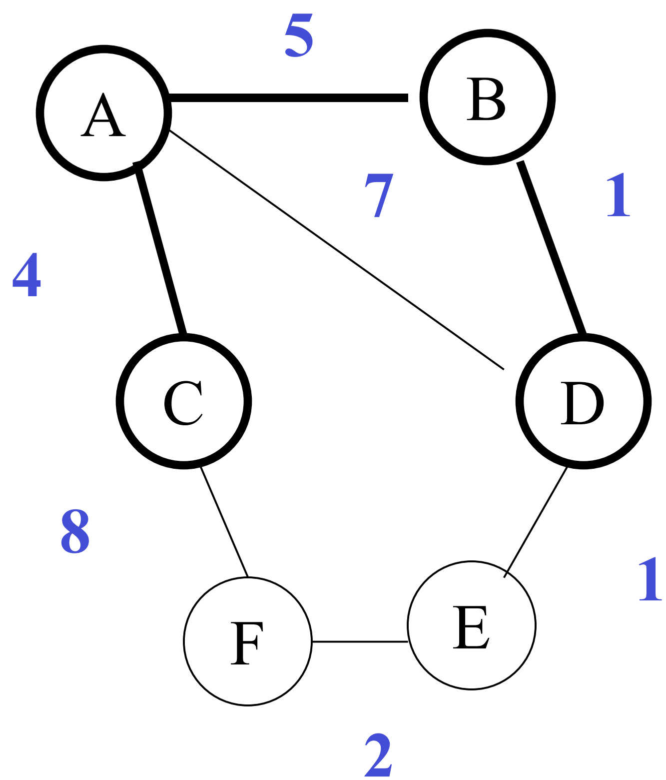
Example

Node	Status	Link	Weight
A	Tree	--	0
B	Tree	A	5
C	Tree	A	4
D	Fringe	B	1
E			
F	Fringe	C	8



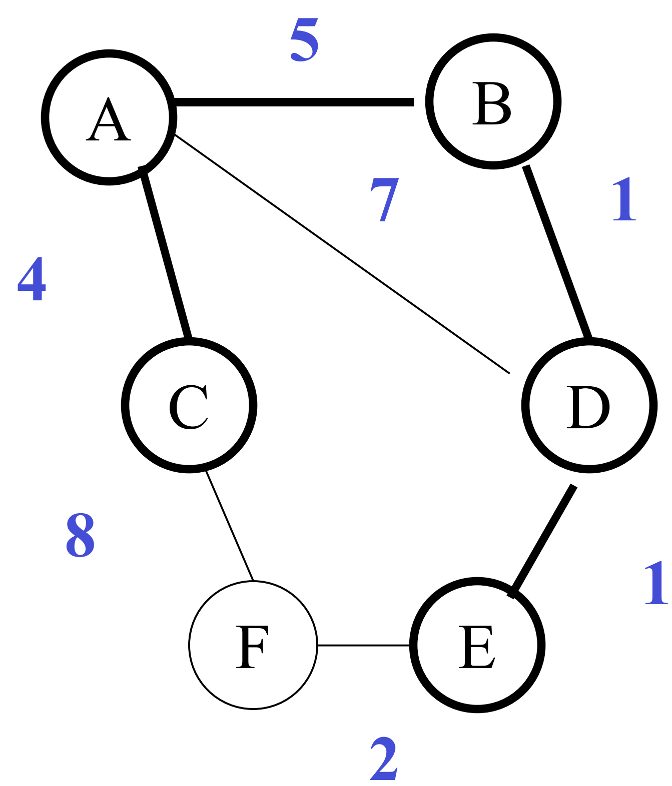
Example

Node	Status	Link	Weight
A	Tree	--	0
B	Tree	A	5
C	Tree	A	4
D	Tree	A	6
E	Fringe	D	1
F	Fringe	C	8



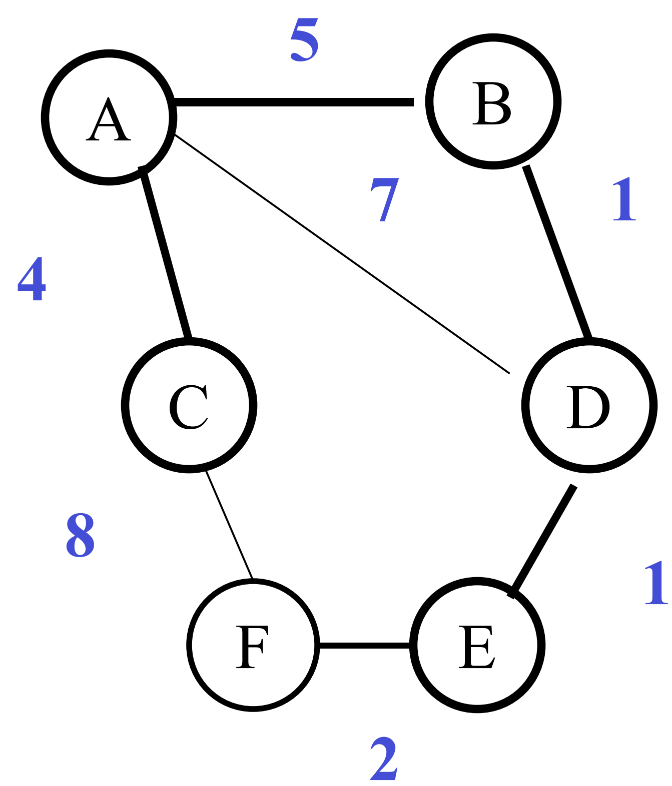
Example

Node	Status	Link	Weight
A	Tree	--	0
B	Tree	A	5
C	Tree	A	4
D	Tree	A	6
E	Tree	D	1
F	Fringe	E	2



Example

Node	Status	Link	Distance
A	Tree	--	0
B	Tree	A	5
C	Tree	A	4
D	Tree	B	1
E	Tree	D	1
F	Tree	E	2



Sorting

- **Sorting is important**
  - **Can search sorted data in O(log n) time**
- **There are many different sorting algorithms**
- **In this class, we will look at 5**
  - **Insertion**
  - **Quick**
  - **Merge**
  - **Heap**
  - **Radix**

Why study more than one?

- Each algorithm has strengths & weaknesses
  - No one best for all situations
- Good examples of array algorithms
- Good example of many algorithms for the same task

2-array vs in-place sorting

- **Simplest to describe: 2-array**
  - Given: array A, not in order
  - Produce: array B, same numbers but in order
- **More efficient use of memory: In-Place**
  - Given: array A, unsorted
  - Produce: array A, same numbers but in order

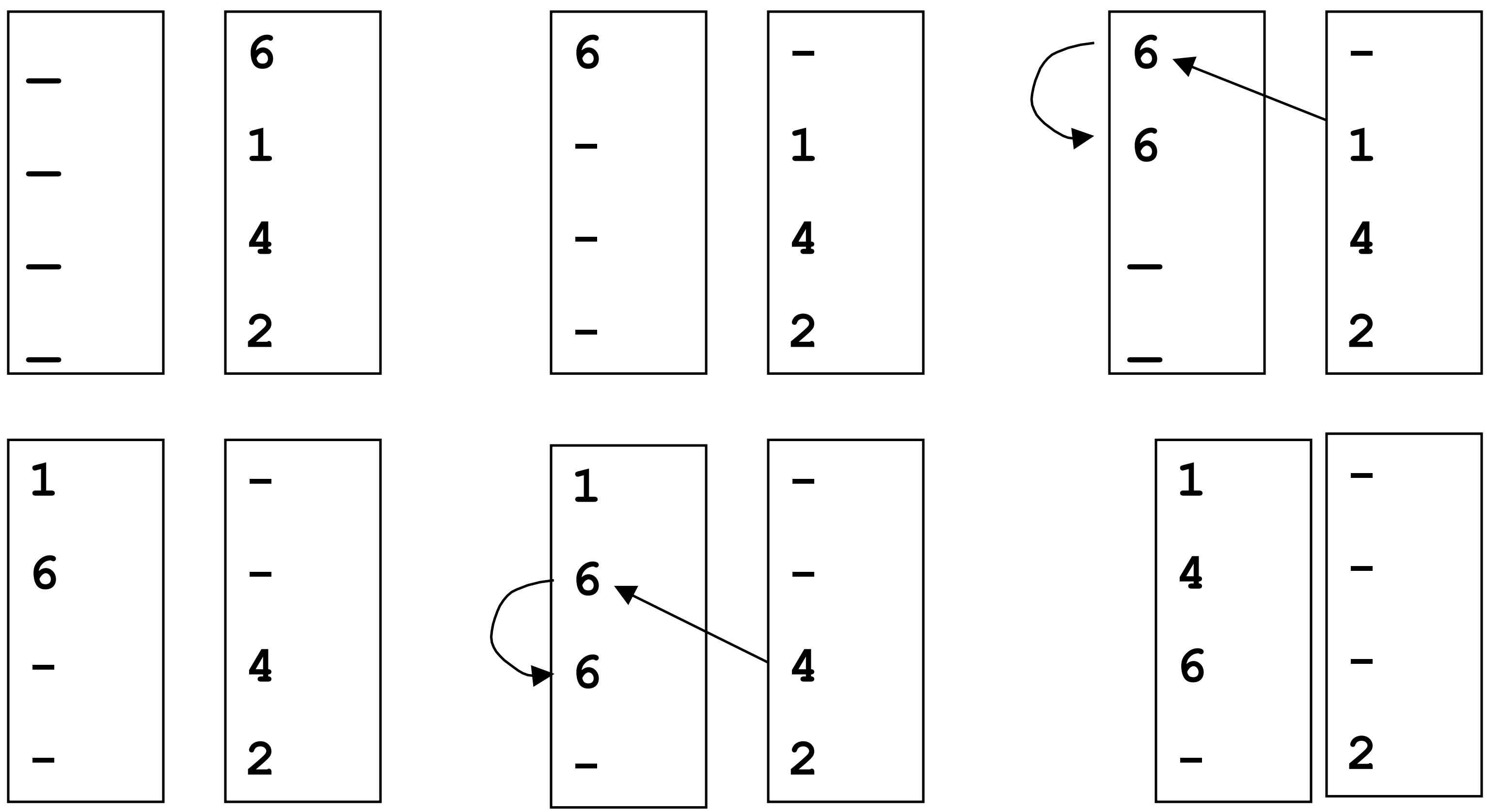
General In-Place

- **In-Place: extra memory constant as input size grows**
  - **O(1)**

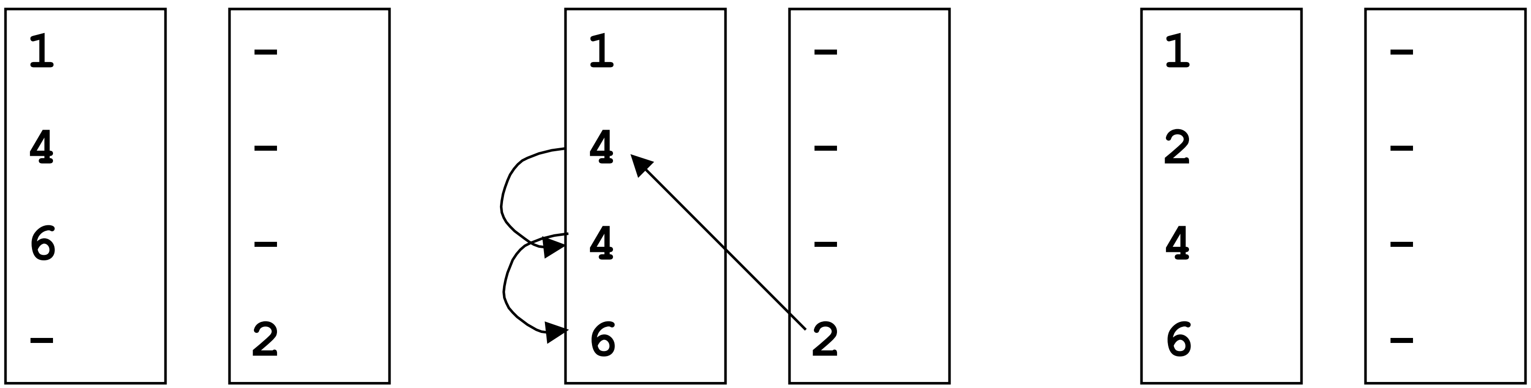
Insertion Sort

- **To sort: take numbers one by one from unsorted and insert in order in sorted**

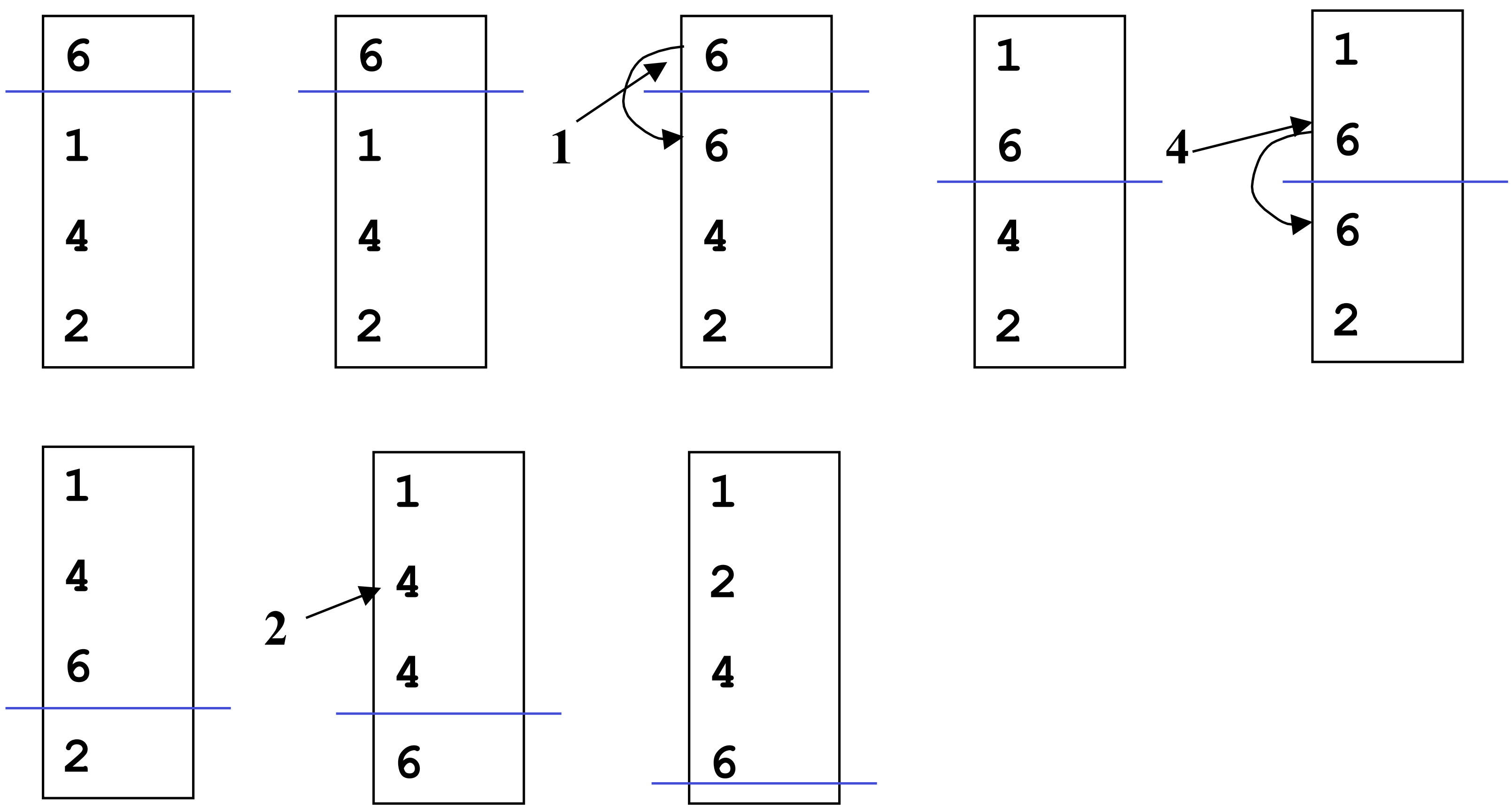
Insertion Sort



Insertion Sort (cont.)



In-Place Insertion Sort



Insertion Sort

- How fast is insertion sort
- Count: comparison of two numbers to be sorted
  - 1st insertion: 0 compares
  - 2nd insertion 1 compare
  - 3rd insertion 2 compares worst case
  - … …
  - nth insertion n-1 compares worst case

Insertion Sort

- **$0+1+2+\ldots+n-1 = (n-1)*((n-1)+1)/2 = O(n^2)$**
- **Also cost of moving lots of data**

Code

- **See http://www.cs.ubc.ca/~harrison/Java/InsertionSortAlgorithm.java.html**

Divide and Conquer

- General approach when > O(n)
  - divide data in half
  - process each half
  - combine results

N log N Sorts

- **Quicksort:**
  - **Partition**
    - **Split data into two groups, all in one group < any in other group**
  - **sort groups separately**
    - **use quicksort recursively**
  - **append**
    - **if partition & sort are in-place this is a no-op**

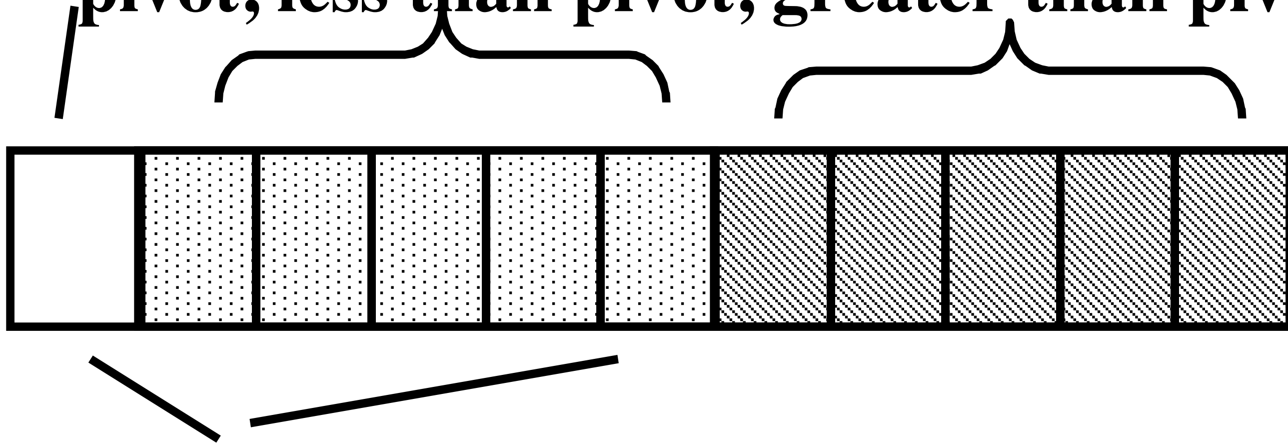
Partition

- Choose a “pivot” value from data
  - ideal would be median => equal size lists
  - but takes too long to find median
  - simplest: pivot = first
    - but in order -> worst case
  - safer: median of 1st, last, middle

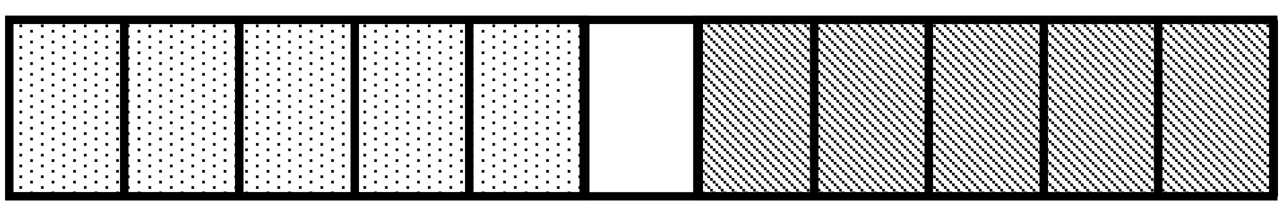
Partition

- **Trick: first partition like this**

pivot, less than pivot, greater than pivot



then swap



Partition

- Use 2 pointers: left and right
  - move left from low+1 up until A[left] > pivot
  - move right from high down until A[right]<pivot
  - Swap
  - Repeat until left>=right

Quicksort

- **How sort regions left & right of pivot?**
  - **Quicksort! (unless nothing in region)**
    - **Actually, insertion sort faster for small regions**
      - **size<10 or so**

Complexity

- Partition takes O(n) time where n is the number of numbers to partition
- Best case: assume partition always into equal halves
  - Suppose 15 numbers in array
  - partition 0 - 14 15 compares
  - partition 0-6, 8-14 7+7=14 compares

    … … always O(n) compares
- Each level divides partition size by 2, stop at size 1
  - log n levels
- Total: O(n log n)

Complexity

- **Worst case: always divide into 0 and all-but pivot**
  - **15 -> 14 -> 13 -> … 1: O(n) levels, total $O(n^2)$**
- **Average case: O(n logn) like best**

Code

- See http://www.cs.ubc.ca/~harrison/Java/QSortAlgorithm.java.html

Merge sort

- Divide & Conquer:
  - split in two parts
    - no comparisons done in split
  - sort each part
  - merge the parts
- Cf quicksort which does comparisons in split and not in combine

Merge

Combine 2 sorted lists into one big sorted list

- compare smallest remaining in each list
- move smallest to output
- when one list empty move all of other list
- Needs extra space
  - linked lists or second array

Complexity

- Merge takes O(n) where n is size of result
- Like quicksort, level i does 2^i sublists, each of length $O(N/2^i)$ => O(N) work at each level
- Best, worst, average all do O(log n) levels
- Complexity is O(n log n)

Code

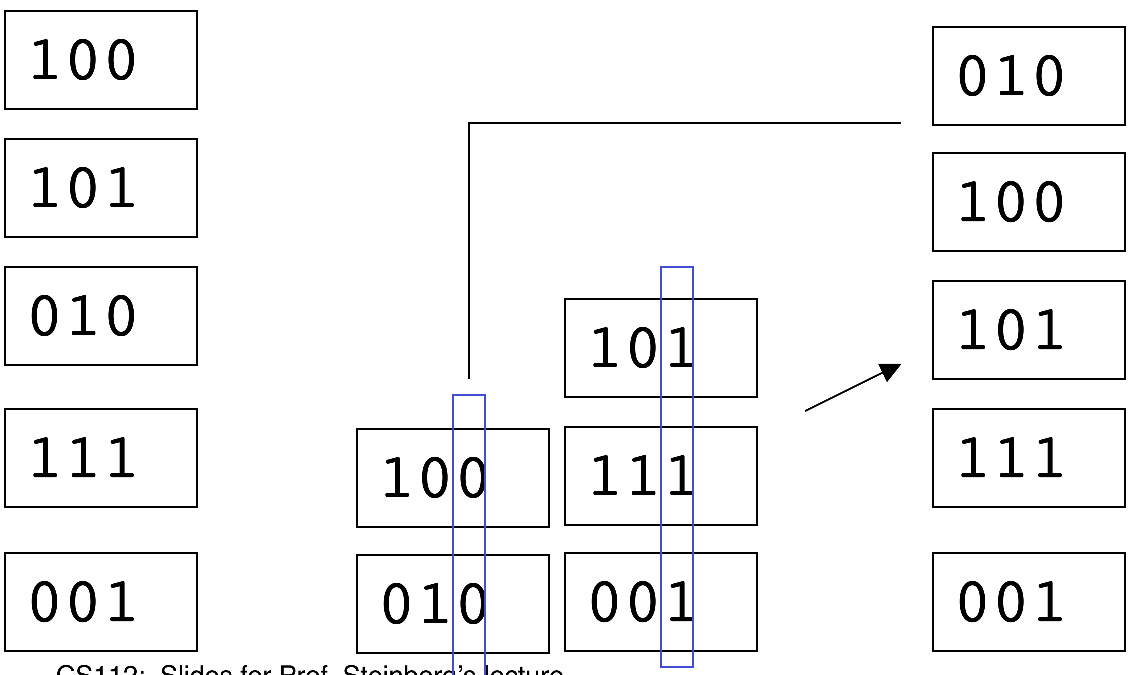
- **http://www.cs.ubc.ca/~harrison/Java/ExtraStorageMergeSortAlgorithm.java.html**

Merge vs Quick

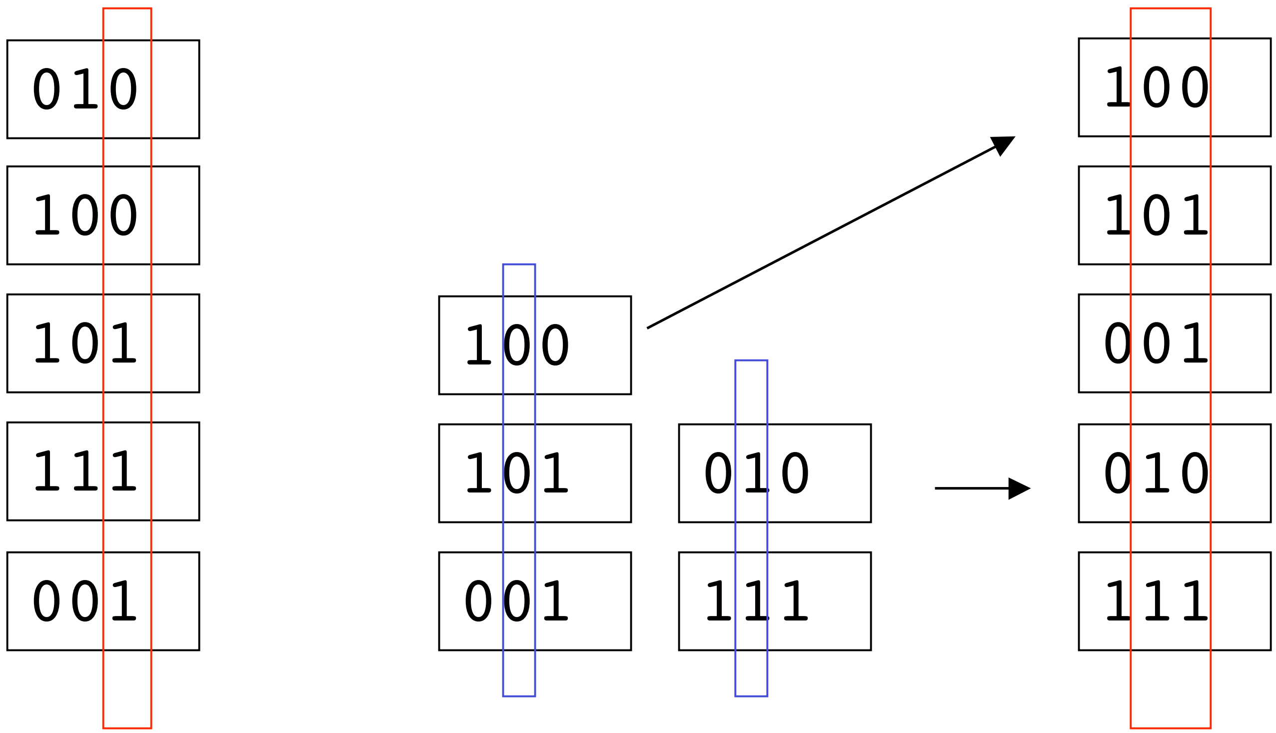
- **Merge has space overhead and also time overhead but even worst case is O(n log n)**
- **Quick is in-place and low time overhead but (very unlikely) worst case $O(n^2)$**

Radix sort

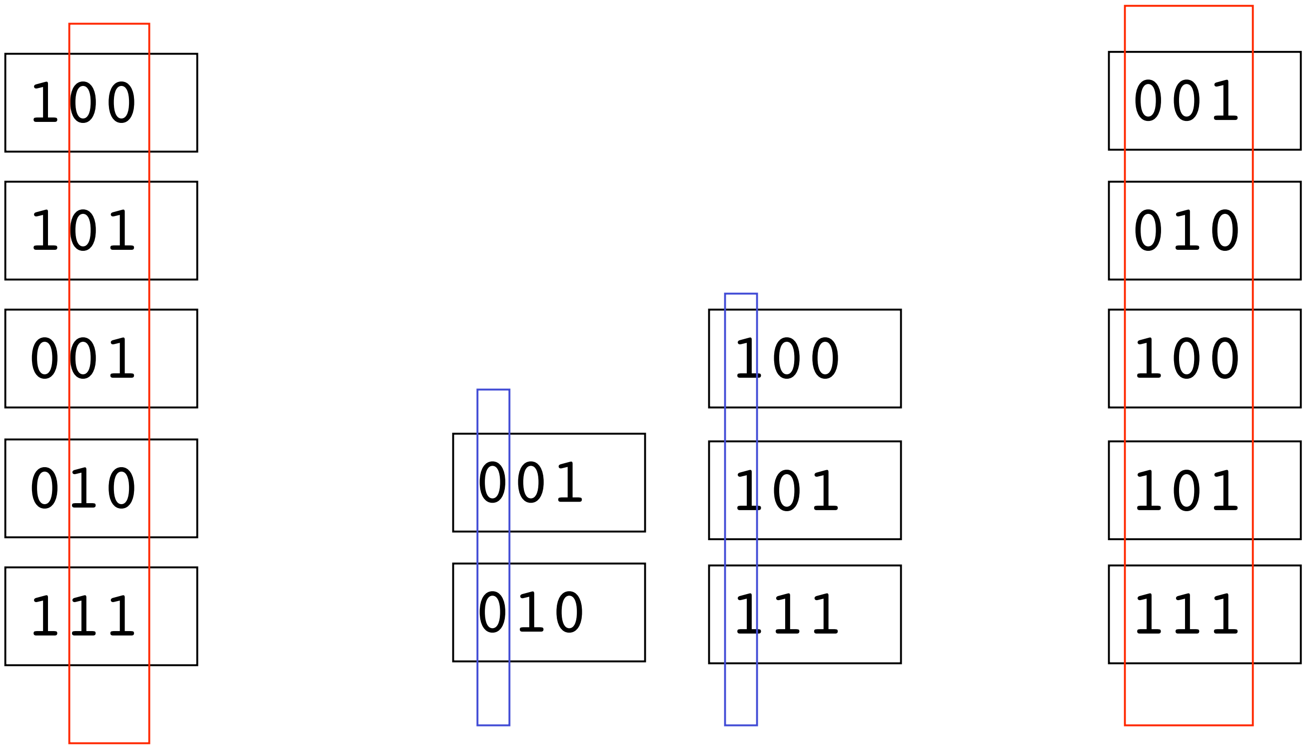
- Put in piles by last digit
- collect piles in order
- put in piles by next-to-last digit …



Radix Sort



Radix Sort



Complexity of Radix Sort

- Outer loop: once per digit
  - Inner loop: once per number
- Compares: d * n
  - If we consider d as a constant, O(n)