COMP2230 Introduction to Algorithmics

Lecture 6

Professor Ljiljana Brankovic

Lecture Overview

Searching Graphs

Divide and Conquer

Sorting Algorithms

Next week: Greedy Algorithms

Searching graphs

- The problems we discussed last week were just examples of searching graphs (topological sort, backtracking, games, etc.)

- Goal: visit all nodes in the graph

- Two basic strategies:
  - **Depth first search**
    - every time you see a new node, stop and go and look at that node... You will end up far away from "home" node - brave heart approach
  - **Breadth first search**
    - look at all the nearest nodes, in the order you found them, before expanding out.... Always stay close to the "home" node - cautious approach

Both work for directed and undirected graphs.

Depth-First Search

G = (V,E)

Graph can be directed or undirected

Each node marked *visited* or *not-visited*

Time complexity $\Theta(|V| + |E|)$; Why?

Strictly speaking, the complexity will depend on data structure used to represent the graph

procedure dfSearch(G)
  for each $v \in V$ do
    visit[v] ← false
  for each $v \in V$ do
    if !visit [v] **then** *dfs(v)*

procedure *dfs(v)*
  visit[v] ← true
  for each node *w* adjacent to *v*
    if !visit[w] **then** *dfs(w)*

Depth-First Search

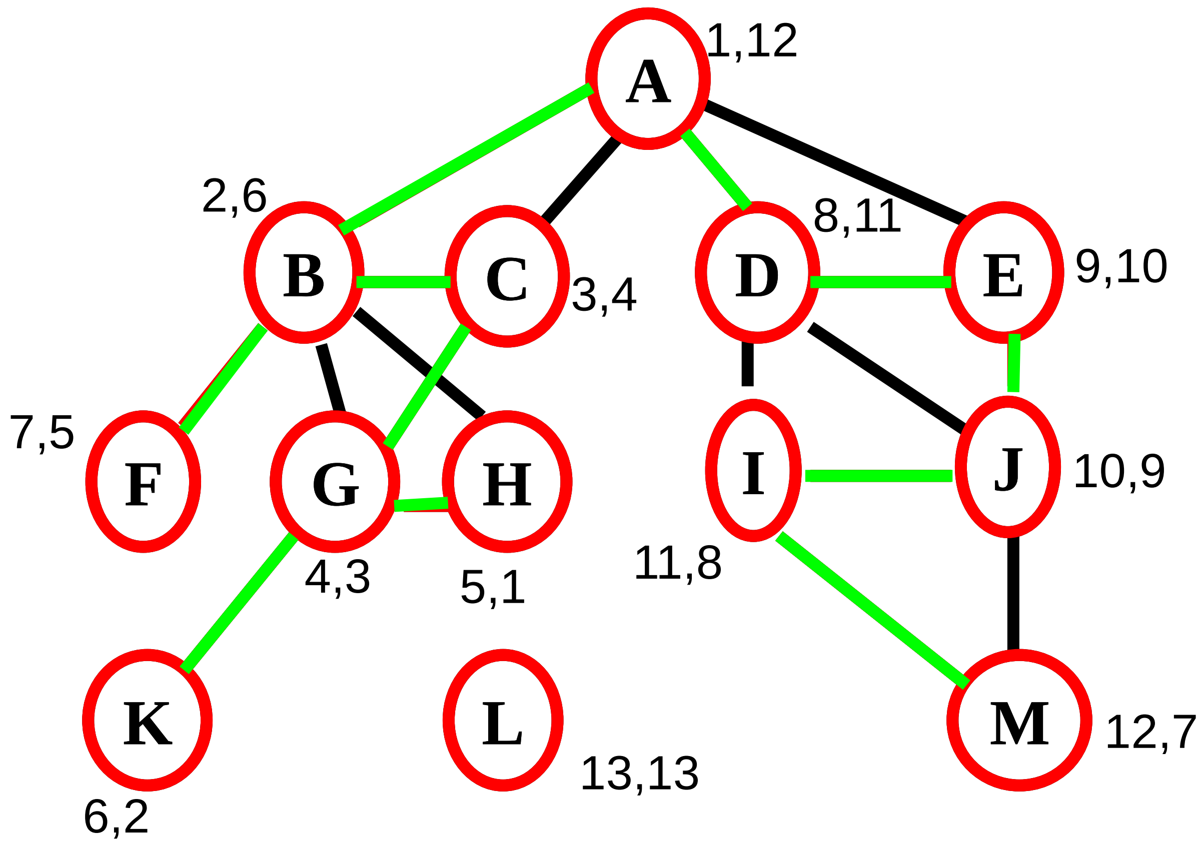
DFS gives two different orderings of nodes:

1. the order in which they are visited, and
2. the order in which they become a dead end.

```
procedure dfSearch(G)
    count1 ← 0
    count2 ← 0
    for each v ∈ V do
        visit1[v] ← 0
    for each v ∈ V do
        if visit1[v] = 0
        then dfs(v)
```

```
procedure dfs(v)
    count1 ← count1 + 1
    visit1[v] ← count1
    for each node w adjacent to v
        if visit1[w] = 0 then dfs(w)
    count2 ← count2 + 1
    visit2[v] ← count2
```

Example 1- DFS example



The black edges are called "back" edges and they connect a vertex with its ancestors in the DFS tree.

Data structures?

- -stack
- -adjacency lists or adjacency matrix

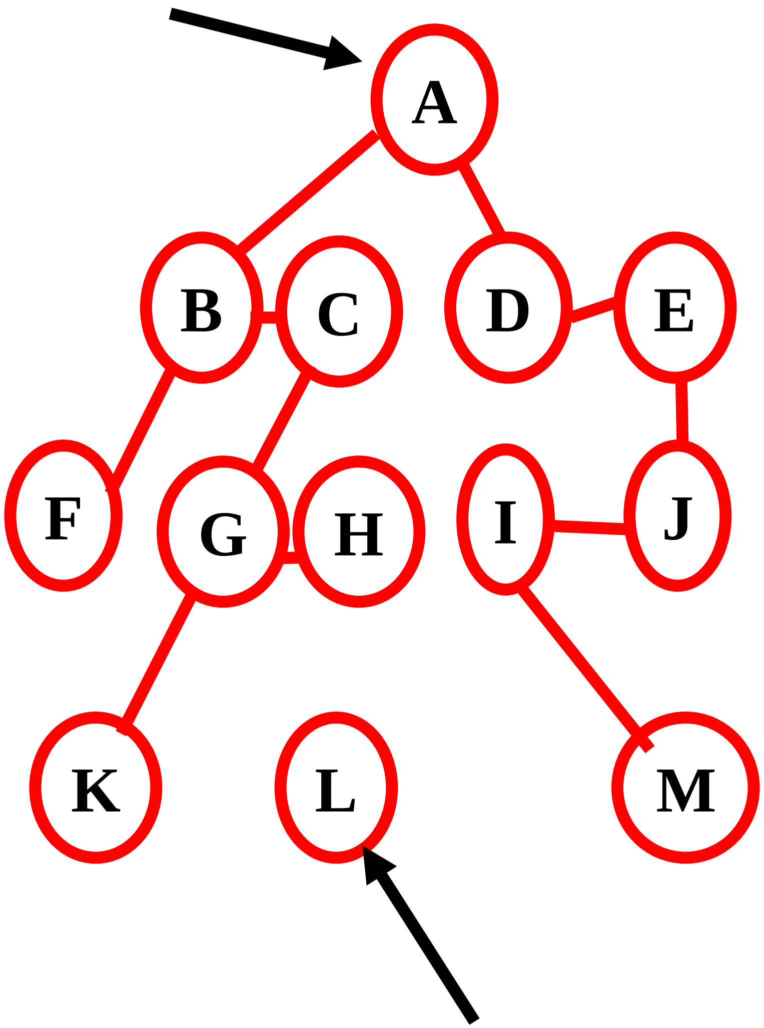
Algorithm 4.2.2 Depth-First Search

This algorithm executes a depth-first search beginning at vertex `start` in a graph with vertices `1,…,n` and outputs the vertices in the order in which they are visited. The graph is represented using adjacency lists; *adj[i]* is a reference to the first node in a linked list of nodes representing the vertices adjacent to vertex *i*. Each node has members *ver*, the vertex adjacent to *i*, and *next*, the next node in the linked list or `null`, for the last node in the linked list. To track visited vertices, the algorithm uses an array *visit*; *visit[i]* is set to `true` if vertex *i* has been visited or to `false` if vertex *i* has not been visited.

```
Input Parameters: adj
Output Parameters: None
dfs(adj,start)  {
    n = adj.last
    for i = 1 to n
        visit[i] = false
   for i = 1 to n
       if (!visit[i])
          dfs_recurs(adj,i)
}

dfs_recurs(adj,i) {
    println(i)
    visit[i] = true
    trav = adj[i]
    while (trav != null) {
        i = trav.ver
        if (!visit[i])
            dfs_recurs(adj,i)
        trav = trav.next
    }
}
```

DFS - tree



Tree is not necessarily binary

DFS is often used for:

- spanning trees (not minimum)
- finding connected components
- analyzing graph structure (is graph acyclic?)
- finding cut vertices (articulation points)
- topological sorting
- backtracking

Breadth-First Search

- **DFS**
  - visits the neighbour of neighbour of...
  - naturally recursive
  - uses stack (possibly implicit) to order nodes
  - LIFO
- **BFS**
  - visits all the neighbours before advancing
  - not naturally recursive
  - uses a queue of nodes
  - FIFO

Breadth-First Search

G = (V,E)

Graph can be directed or undirected

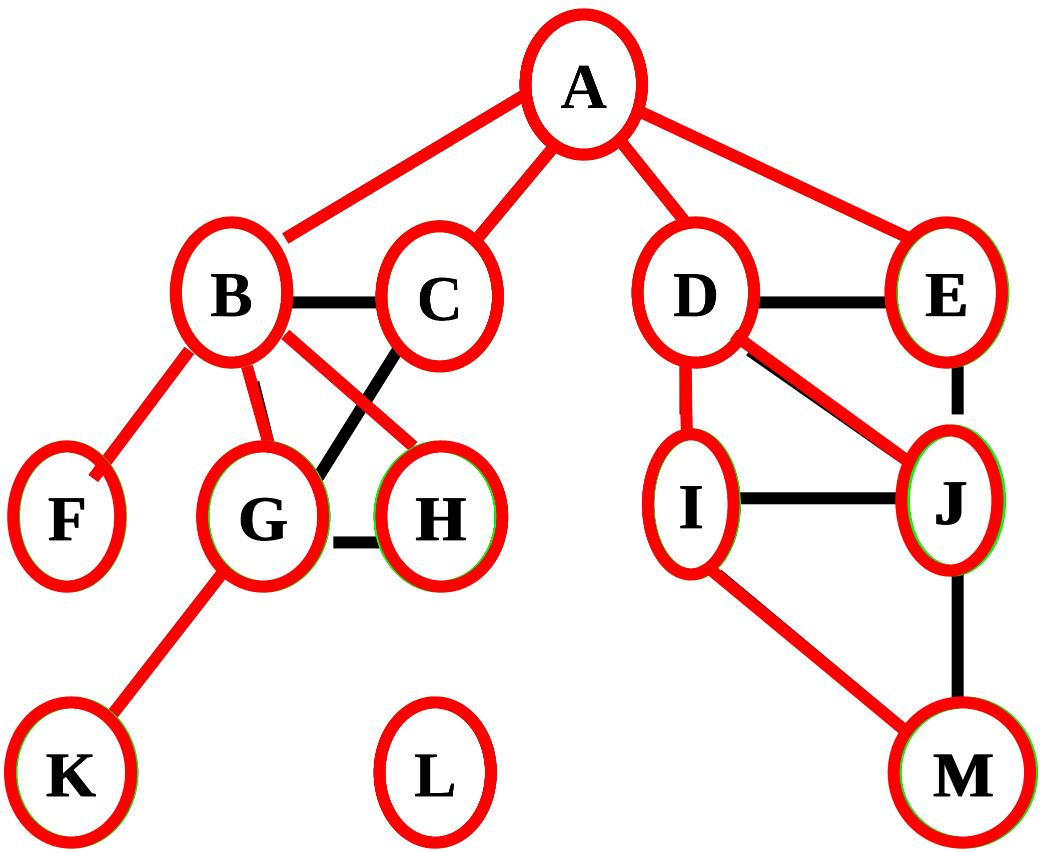
Each node marked *visited* or *not-visited*

Time complexity is $\Theta(|V| + |E|)$. Why?

```
procedure bfSearch(G)
  for each v ∈ V do
    visit[v] ← false
  for each v ∈ V do
    if !visit[v] then
      bfs(v)
```

```
procedure bfs(G)
    Q ← empty-queue
    visit[v] ← true
    enqueue v onto Q
  while Q not empty
      u ← Q.dequeue
    for each node w adjacent to u
      if !visit[w] then
          visit[w] ← true
          enqueue w
```

Example 2 - BFS example



The black edges are called "cross" edges and they connect vertices on the same or adjacent levels in the BFS trees.

Algorithm 4.3.2 Breadth-First Search

This algorithm executes a breadth-first search beginning at vertex *start* in a graph with vertices *1,...,n* and outputs the vertices in the order in which they are visited.

The graph is represented using adjacency lists; *adj[i]* is a reference to the first node in a linked list of nodes representing the vertices adjacent to vertex *i*. Each node has members *ver*, the vertex adjacent to *i*, and *next*, a reference to the next node in the linked list or `null`, for the last node in the linked list.

Algorithm 4.3.2 Breadth-First Search

To track visited vertices, the algorithm uses an array *visit*; `visit[i]` is set to `true` if vertex i has been visited or to `false` if vertex i has not been visited. The algorithm uses an initially empty queue q to store pending current vertices.

The expression *q.enqueue(val)* adds *val* to q.

The expression *q.front( )* returns the value at the front of q but does not remove it.

The expression *q.dequeue( )* removes the item at the front of q.

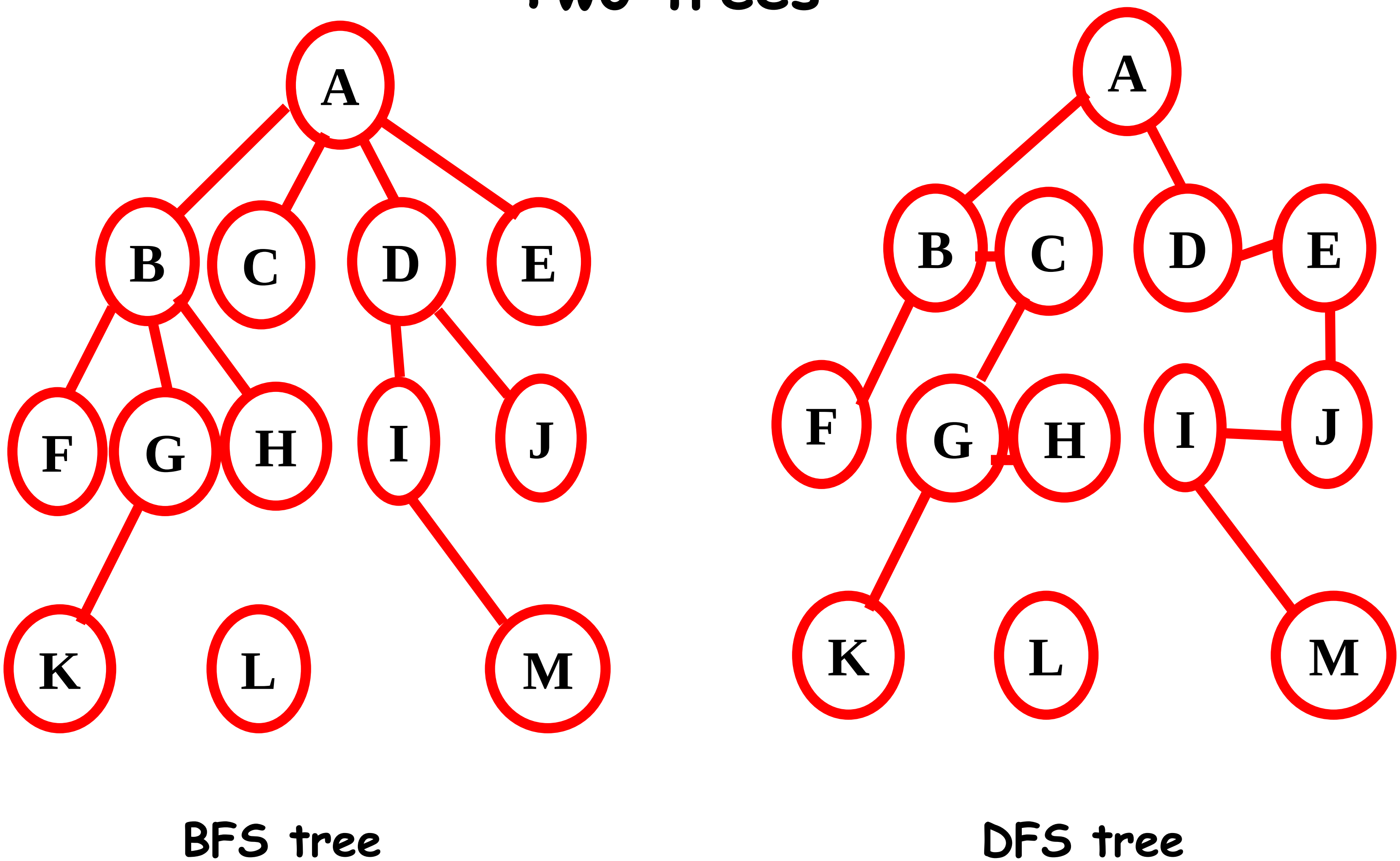
The expression *q.empty( )* returns `true` if q is empty or `false` if q is not empty.

```
Input Parameters: adj,start
Output Parameters: None
bfs(adj,start)  {
    n = adj.last
    for i = 1 to n
        visit[i] = false
    visit[start] = true
    println(start)
    q.enqueue(start) // q is an initially empty queue
    while (!q.empty()) {
        current = q.front()
        q.dequeue()
        trav = adj[current]
        while (trav != null) {
            v = trav.ver
            if (!visit[v]) {
                visit[v] = true; println(v); q.enqueue(v)
            }
            trav = trav.next
        }
    }
}
```

Homework

Modify this algorithm so that it also works for a disconnected graph.

Two trees

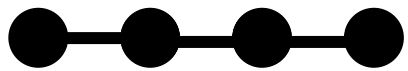
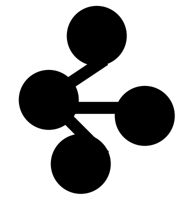


Compare edges coming from node A

Uses for Breadth First Search

- Constructing spanning trees (similar to DFS)
- Finding connected components (similar to DFS)
- Partial exploration of large graphs
- Consider a problem involving exploration of a graph
- Why would we choose BFS over DFS?

BFS vs DFS

- DFS rushes deep into graph, before even exploring nearby options.
- BFS visits nearby neighbours before going deeper.

- Consider a game (like chess):
  - BFS considers all options 1 move ahead before moving
  - DFS makes a move, then another...

Example 3 - BFS graph colouring

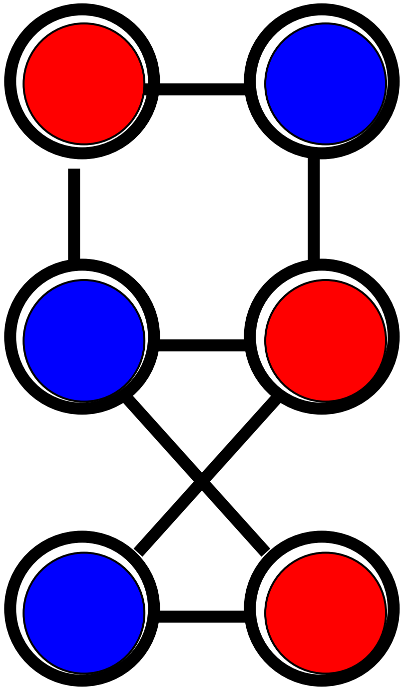
- Graph/map colouring

  A crazed Knights fan hires you to paint every room in their house alternatingly red & blue. They don't want a red room next to another red room nor a blue room next to another blue room.

  - How could we represent this as a graph?

  - How could we test if it was possible to paint the house as desired and if so, which rooms to paint which colour?

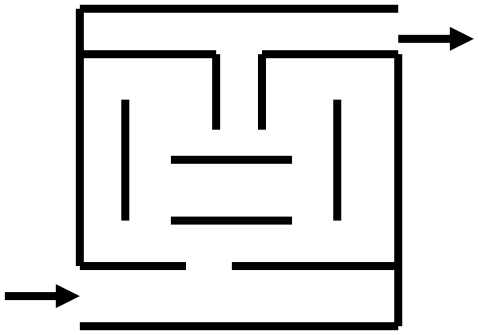
Example 3 - BFS graph colouring



- Run BFS
  - include "colour" variable (could be Boolean)
  - Paint 1st room red (or blue)
  - All nodes reached by BFS in each iteration are painted opposite colour to previous iteration
- If we find a node which is already coloured, it must be opposite to the current colour

- Would DFS work?

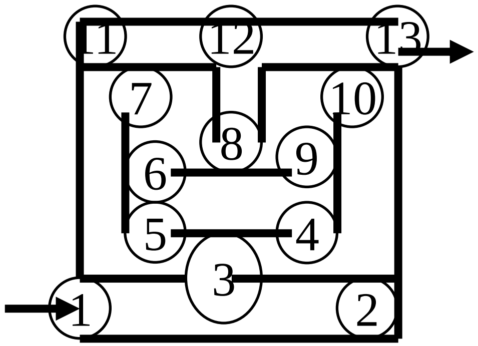
Example 4 - MAZE

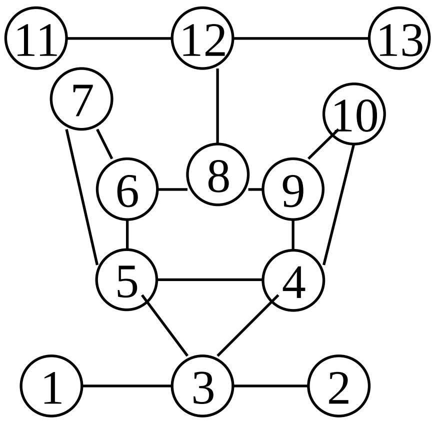


We shall model a maze as an undirected graph, where vertices will be used to represent

- starting point,
- finishing point,
- dead ends, and
- all points where more than on path can be taken.

Example 4 - MAZE





- Would you use DFS or BFS to get yourself out of the maze? Why?

BFS vs DFS

	DFS	BFS
Data structures	•Stack •Adjacency lists or adjacency matrix	•Queue •Adjacency lists or adjacency matrix
Complexity	$\Theta(\|V\| + \|E\|)$ for adj. lists $\Theta(\|v\|^2)$ for adj. matrix	$\Theta(\|V\| + \|E\|)$ for adj. lists $\Theta(\|v\|^2)$ for adj. matrix
Applications	Finding spanning trees, connected components, cut vertices (articulation points) Exploring graphs Topological sort Backtracking	Finding spanning trees connected components shortest path Exploring graphs Web crawling Social networking Garbage collection Puzzles Games

Divide and Conquer

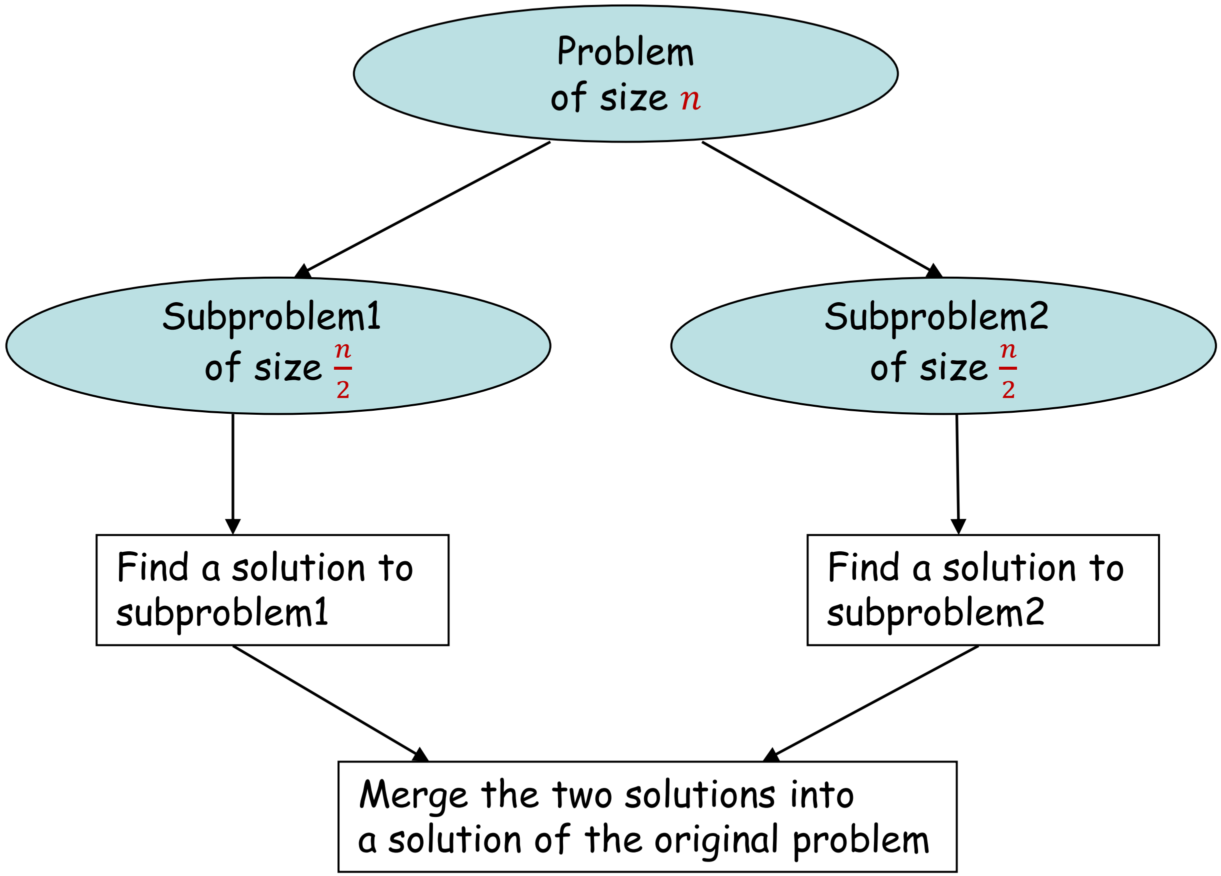
Divide and Conquer is a basic algorithmic design technique that uses recursion to split large problems into smaller sub-problems that can be solved easily.

Works with two basic steps:

- If the problem is small enough, solve it.
- Otherwise split the problem into smaller instances of the same problem and recur.

Not every problem can be solved with this technique.

Divide and Conquer



Divide and Conquer

In general, the problem will be divided into b sub-problems of size $\frac{n}{b}$ and a of these problems will need to be solved; $f(\mathrm{n})$ is the time required to merge the solutions to sub-problems into a solution of the original problem.

$$\mathrm{T}(n) = a \times \mathrm{T}\left(\frac{n}{b}\right) + f(\mathrm{n})$$

Divide and Conquer

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

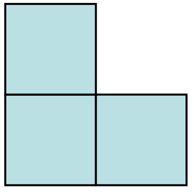
To solve this recurrence relation, we use Master Theorem:

If $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ and $f(\mathrm{n}) = \Theta\left(n^k\right)$ then

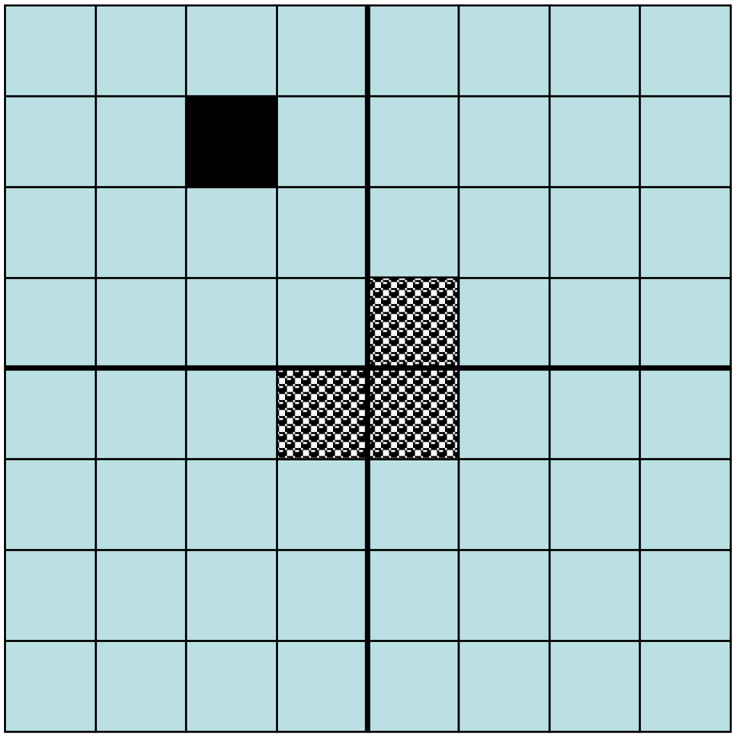
$$T(n) = \begin{cases} \Theta\left(n^k\right) & \text{if } a < b^k \\ \Theta\left(n^k \log n\right) & \text{if } a = b^k \\ \Theta\left(n^{\log_b a}\right) & \text{if } a > b^k \end{cases}$$

Tiling a Deficient Plane

- Imagine we have a set of L shaped tiles and a plane with one square missing.
- How might we use a D & C approach to tile the plane?



Tiling a Deficient Plane



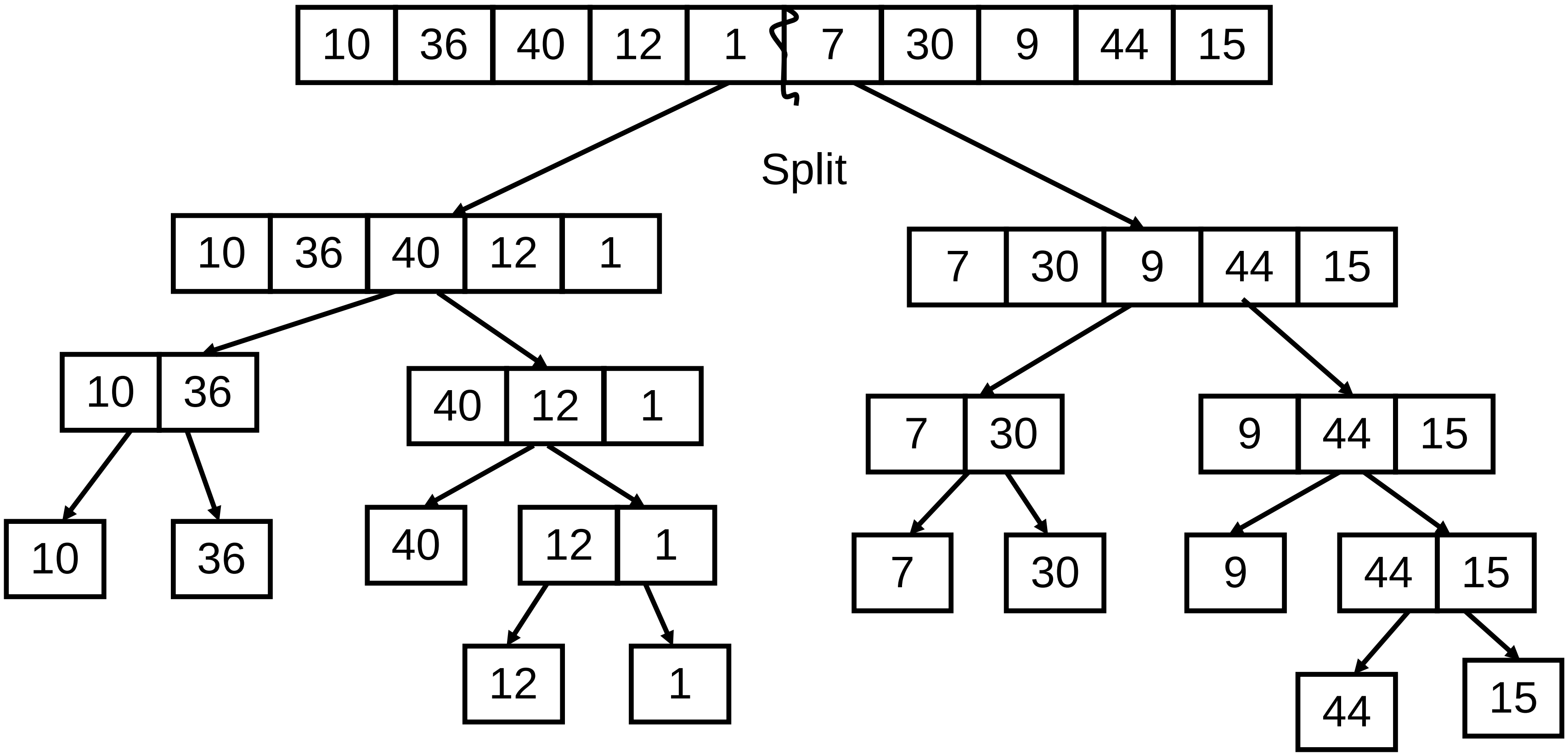
Algorithm 5.1.4 Tiling a Deficient Board with Trominoes

This algorithm constructs a tiling by trominoes of a deficient $n \times n$ board where n is a power of 2.

```
Input Parameters: n, a power of 2 (the board size);
                  the location L of the missing square
Output Parameters: None
tile(n,L) {
    if (n == 2) {
        // the board is a right tromino T
        tile with T
        return
    }
    divide the board into four n/2 × n/2 subboards
    place one tromino as in the previous slide
    // each of the 1 × 1 squares in this tromino
    // is considered as missing
    let m1,m2,m3,m4 be the locations of the missing squares
    tile(n/2,m1)
    tile(n/2,m2)
    tile(n/2,m3)
    tile(n/2,m4)
}
```

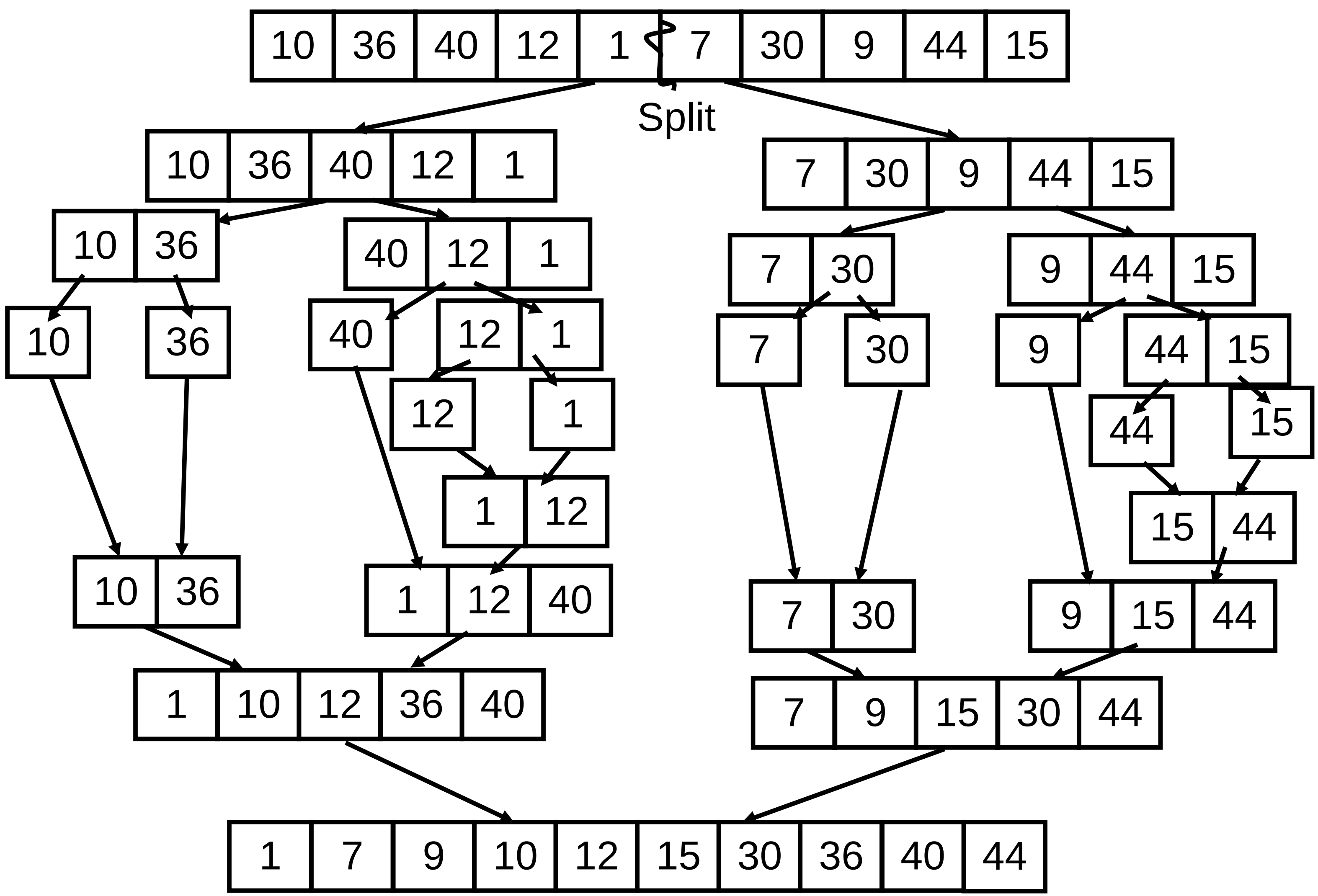
Write the corresponding recurrence relation!

MergeSort



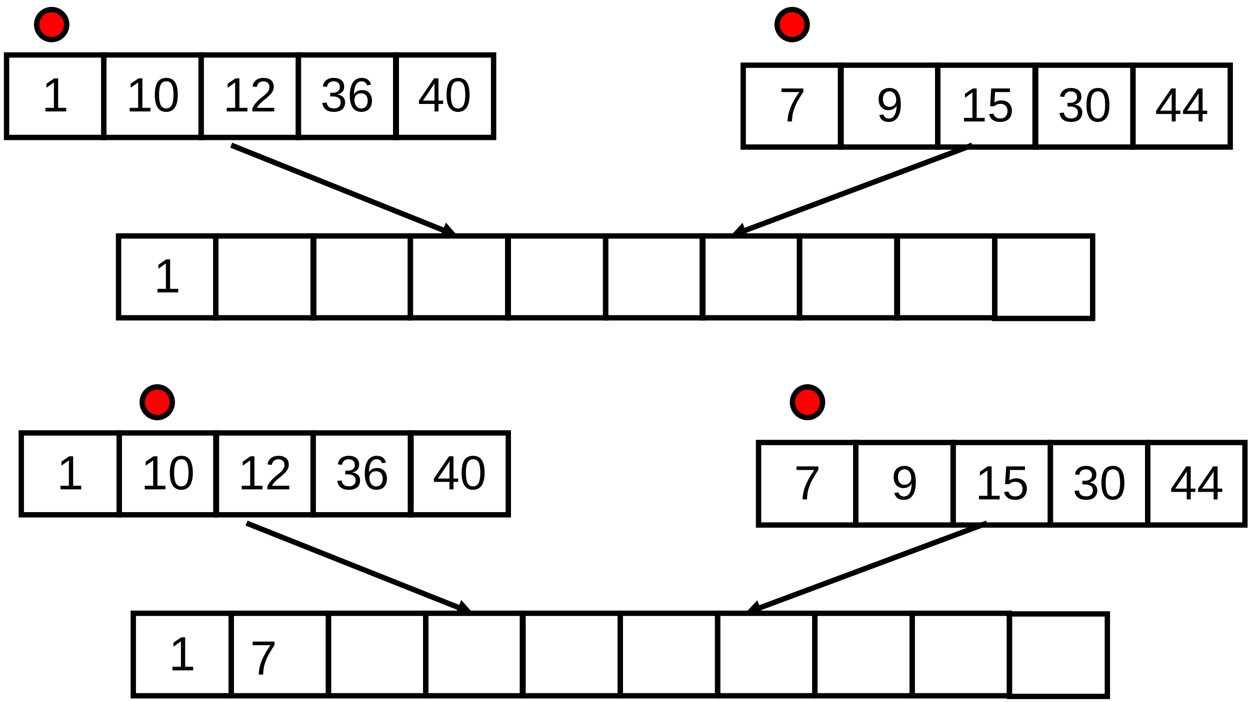
10
36
40
12
1
7
30
9
44
15
Split
10
36
40
12
1
7
30
9
44
15
10
36
40
12
1
7
30
9
44
15
10
36
40
12
1
7
30
9
44
15
12
1
44
15

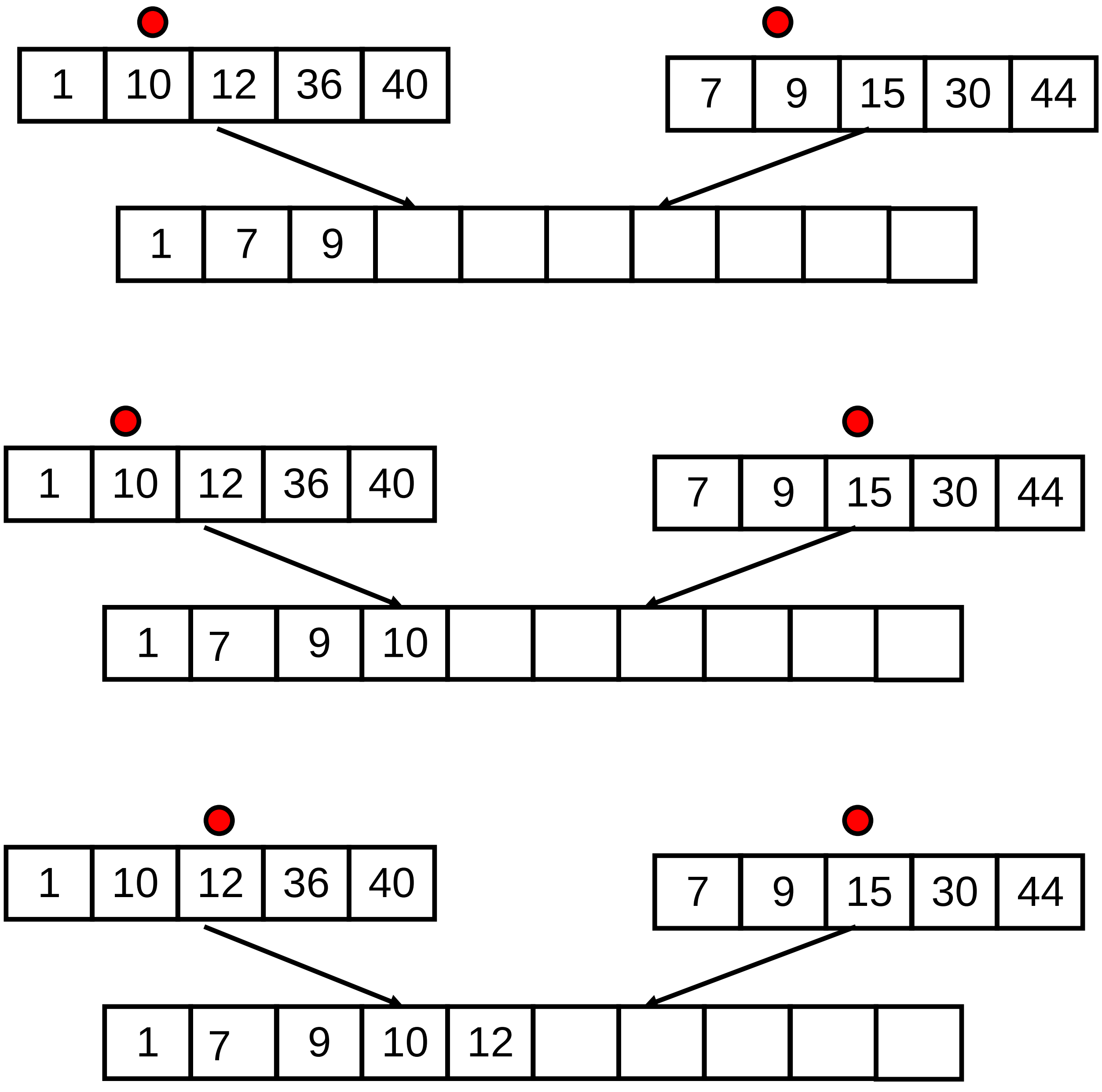
MergeSort

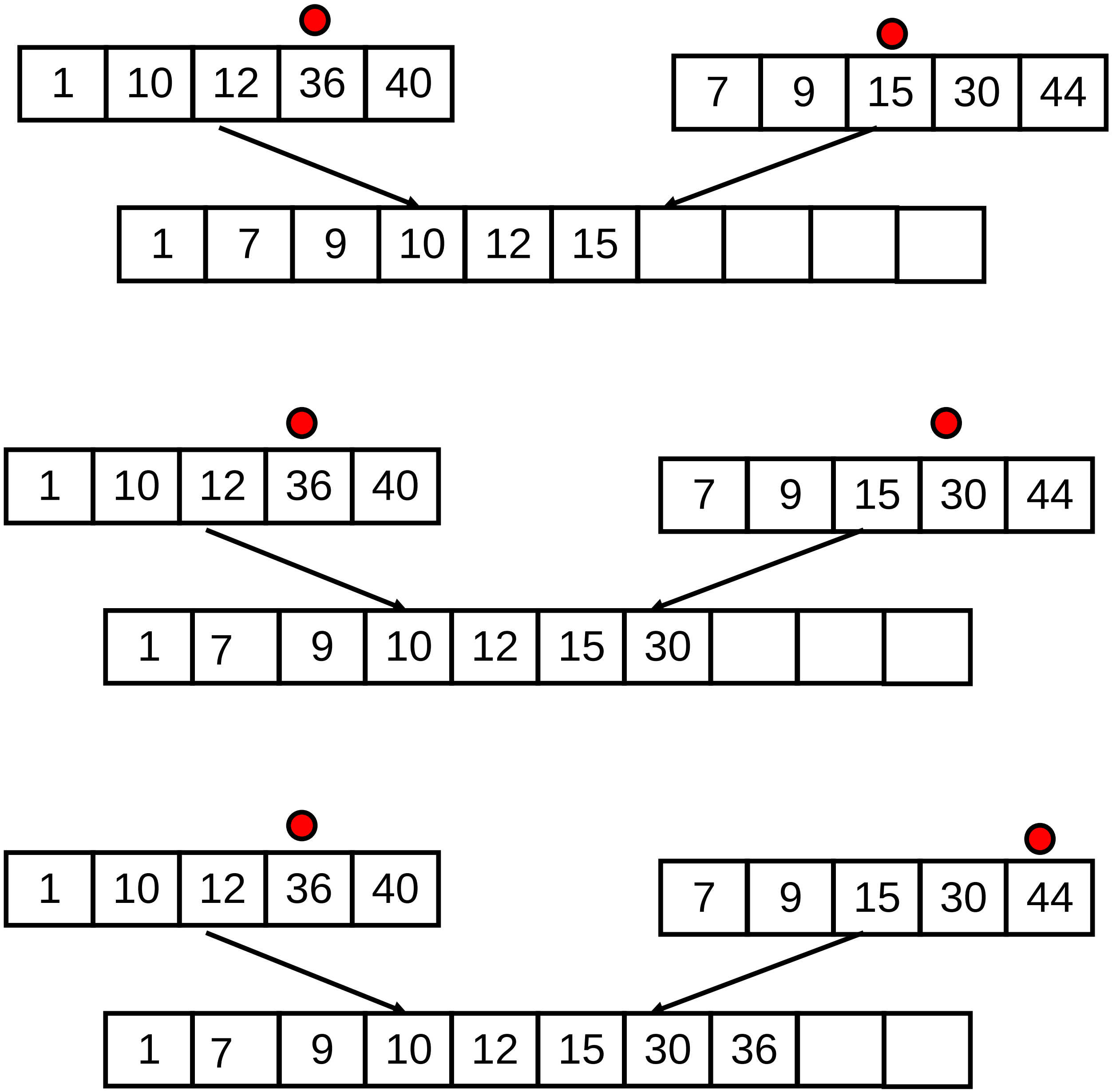


10 36 40 12 1 7 30 9 44 15
Split
10 36 40 12 1
7 30 9 44 15
10 36
40 12 1
7 30
9 44 15
10
36
40
12 1
7
30
9
44 15
12
1
44
15
1 12
15 44
10 36
1 12 40
7 30
9 15 44
1 10 12 36 40
7 9 15 30 44
1 7 9 10 12 15 30 36 40 44

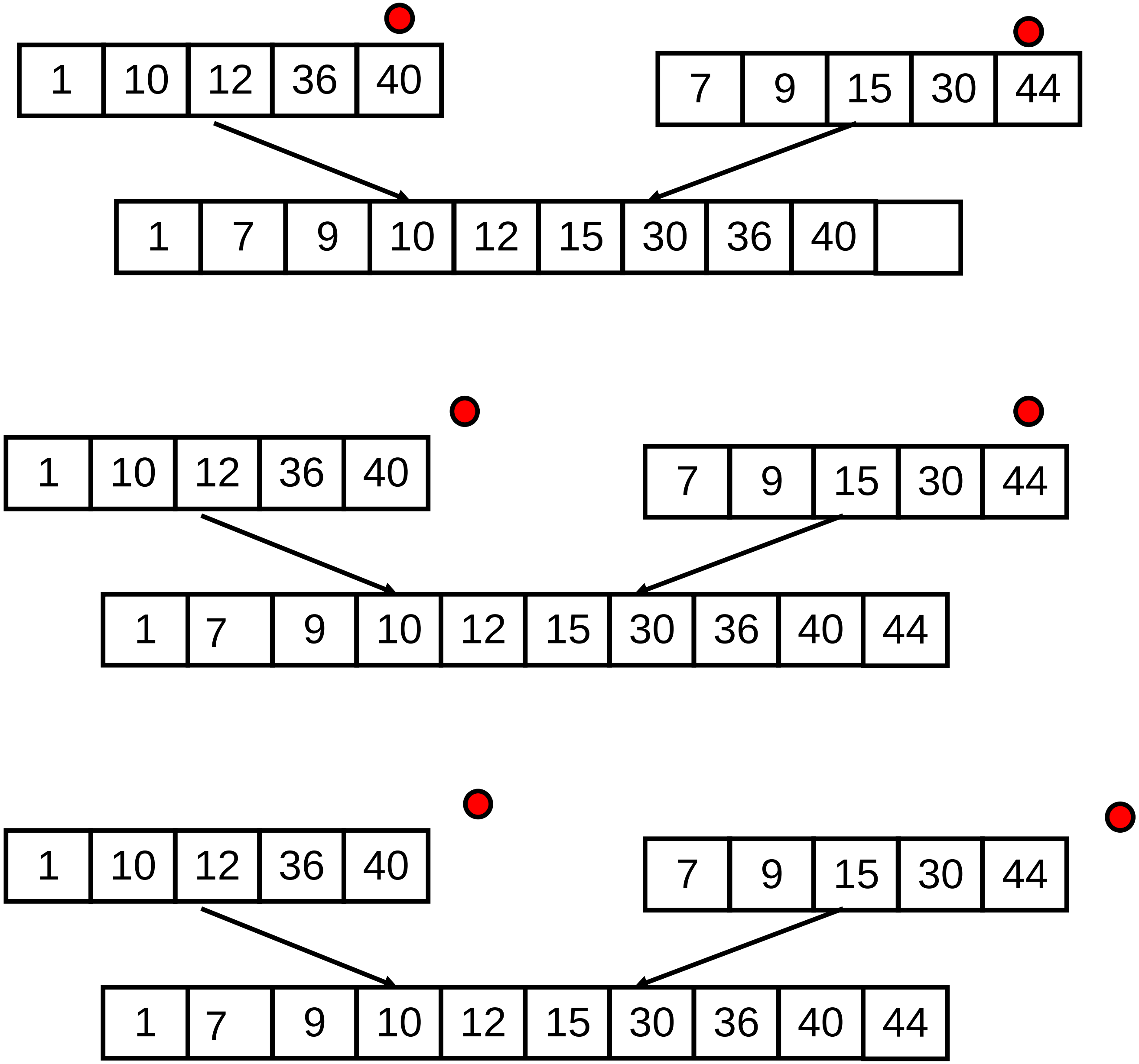
How do we merge?




1 10 12 36 40
7 9 15 30 44
1 7 9
1 10 12 36 40
7 9 15 30 44
1 7 9 10
1 10 12 36 40
7 9 15 30 44
1 7 9 10 12



1 10 12 36 40
7 9 15 30 44
1 7 9 10 12 15
1 10 12 36 40
7 9 15 30 44
1 7 9 10 12 15 30
1 10 12 36 40
7 9 15 30 44
1 7 9 10 12 15 30 36


1 10 12 36 40
7 9 15 30 44
1 7 9 10 12 15 30 36 40
1 10 12 36 40
7 9 15 30 44
1 7 9 10 12 15 30 36 40 44
1 10 12 36 40
7 9 15 30 44
1 7 9 10 12 15 30 36 40 44

Algorithm 5.2.2 Merge

This algorithm receives as input indexes i, m, and j, and an array a, where $a[i],...,a[m]$ and $a[m+1],...,a[j]$ are each sorted in nondecreasing order. These two nondecreasing subarrays are merged into a single nondecreasing array.

```
Input Parameters: a,i,m,j
Output Parameter: a
merge(a,i,m,j) {
    p = i // index in a[i], ... , a[m]
    q = m + 1 // index in a[m + 1], ... , a[j]
    r = i // index in a local array c
    while (p ≤ m && q ≤ j) {
        // copy smaller value to c
        if (a[p] ≤ a[q]) {
            c[r] = a[p]
            p = p + 1
        }
        else {
            c[r] = a[q]
            q = q + 1
        }
        r = r + 1
    }
   ...
```

```
    ...
   // copy remainder, if any, of first subarray to c
    while (p ≤ m) {
        c[r] = a[p]
        p = p + 1
        r = r + 1
    }
    // copy remainder, if any, of second subarray to c
    while (q ≤ j) {
        c[r] = a[q]
        q = q + 1
        r = r + 1
    }
    // copy c back to a
    for r = i to j
        a[r] = c[r]
}
```

Algorithm 5.2.3 Mergesort

This algorithm sorts the array $a[i], \ldots, a[j]$ in nondecreasing order. It uses the merge algorithm (Algorithm 5.2.2).

```
Input Parameters: a,i,j
Output Parameter: a
mergesort(a,i,j) {
    // if only one element, just return
    if (i == j)
        return
    // divide a into two nearly equal parts
    m = (i + j)/2
    // sort each half
    mergesort(a,i,m)
    mergesort(a,m+1,j)
    // merge the two sorted halves
    merge(a,i,m,j)
}
```

Stable Sorting Algorithms

- A sorting algorithm is stable if it preserves the original ordering of equal elements.

Example 5:

Course Code	Last name	. . .
COMP2230	Adamson	
COMP2140	Aston	
COMP2230	Fenn	
COMP2230	Martin	
COMP2410	Peterson	
COMP2410	Smith	
COMP2230	Smith	

- Mergesort is stable.

Bubble Sort

Very, very simple sorting algorithm.

Works by swapping adjacent elements if they're out of order.

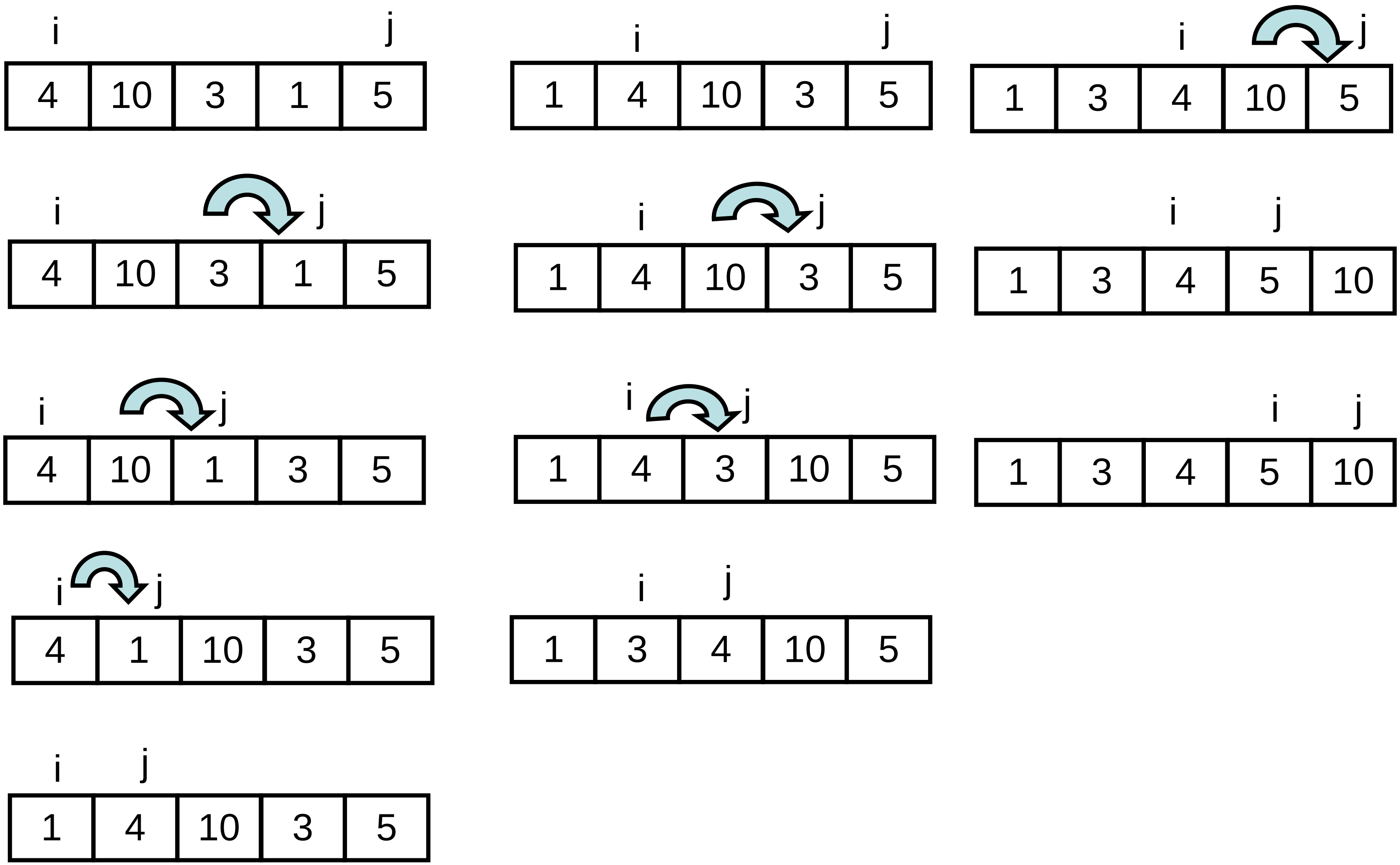
Starts at the end, "bubbling" the lowest element down, incrementally working on smaller sections of the array.

Can work the other way as well, taking the highest to the end.

Bubble Sort

```
bubbleSort(int[] A){
   for (int i = 0; i < A.length; i++){
        for (int j = A.length-1; j > i; j--){
                if (A[j] < A[j-1]){
                        int temp = A[j];
                        A[j] = A[j-1];
                        A[j-1] = temp;
                }
        }
   }
}
```

Bubble Sort


i j
4 10 3 1 5
i j
4 10 3 1 5
i j
4 10 1 3 5
i j
4 1 10 3 5
i j
1 4 10 3 5
i j
1 4 10 3 5
i j
1 4 10 3 5
i j
1 4 3 10 5
i j
1 3 4 10 5
i j
1 3 4 10 5
i j
1 3 4 5 10
i j
1 3 4 5 10

Bubble Sort

Worst case time $\mathrm{T}(n^2)$.

Easy to see from two loops.

What is the worst possible input?

Bubble sort is not commonly used in practice, though it can be sped up through various methods, but this does not change the asymptotic time.

Insertion Sort

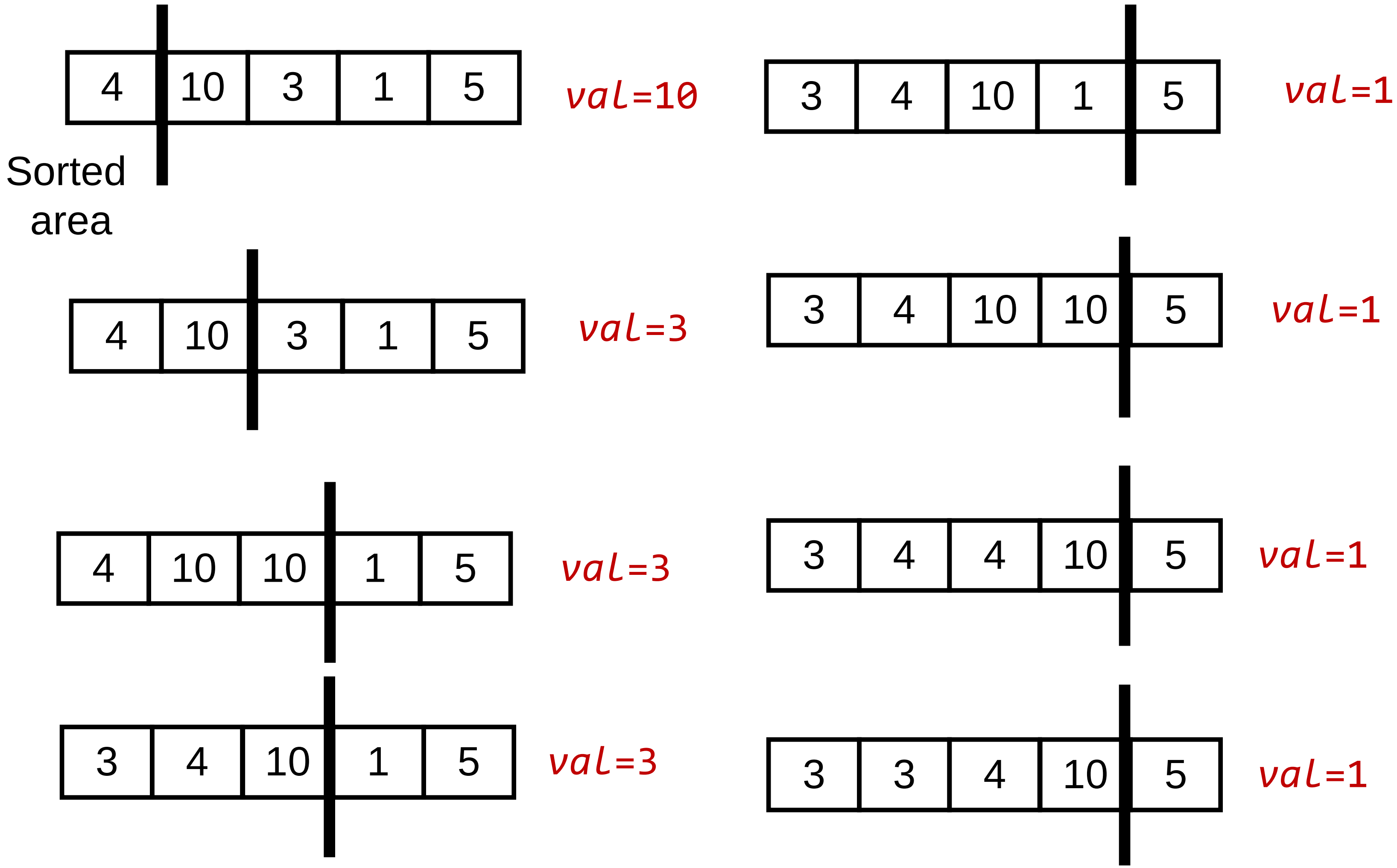
Works by making a “sorted” area at the front of the array (just by first partitioning the first element, which is a sorted one element array), into which it inserts subsequent elements by shuffling elements up until it finds the correct place.

Algorithm 6.1.2 Insertion Sort

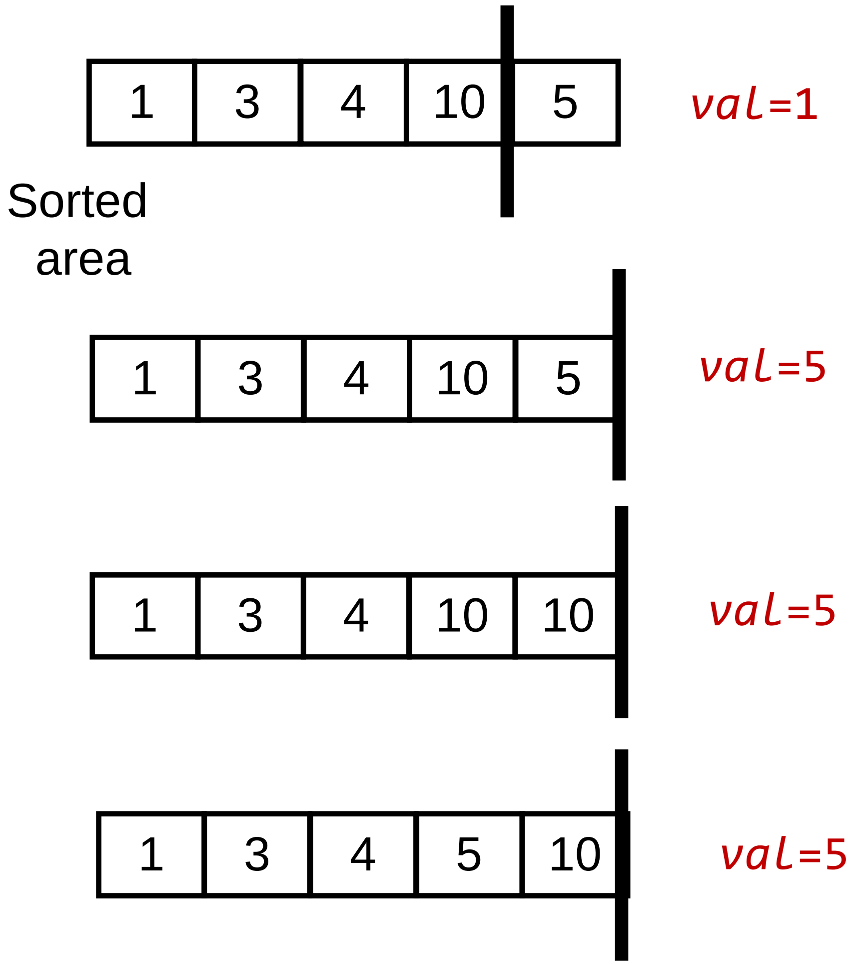
This algorithm sorts the array *a* by first inserting $a[2]$ into the sorted array $a[1]$; next inserting $a[3]$ into the sorted array $a[1]$, $a[2]$; and so on; and finally inserting $a[n]$ into the sorted array $a[1], \ldots, a[n-1]$.

```
Input Parameters: a
Output Parameters: None
insertion_sort(a) {
    n = a.last
    for i = 2 to n {
        val = a[i]     // save a[i] so it can be inserted
        j = i – 1      // into the correct place
        // if val < a[j], move a[j] right to make room
          for a[i]
        while (j ≥ 1 && val < a[j]) {
            a[j + 1] = a[j]
            j = j - 1
        }
        a[j + 1] = val // insert val
    }
}
```

Example 6 - Insertion Sort



Example 6 - Insertion Sort



Insertion Sort

Again, in the worst case, running time is $T(n^2)$.

The worst case is where val has to be inserted in the first position, giving the running time of:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$$

Prove this! That is, prove both O and Ω.

Quick Sort

Quicksort uses a divide and conquer approach, however is very different to merge sort.

First it picks an element to partition the array on, then puts all elements smaller than the partition element to the left of the partition element (but not necessarily in sorted order), and all the larger elements to the right (again not necessarily sorted).

Then it recursively sorts the two sections of the array.

Algorithm 6.2.2 Partition

This algorithm partitions the array

$$a[i], \ldots, a[j]$$

by inserting $val = a[i]$ at the index h where it would be if the array was sorted. When the algorithm concludes, values at indexes less than h are less than val, and values at indexes greater than h are greater than or equal to val. The algorithm returns the index h.

```
Input Parameters: a,i,j
Output Parameters: i
partition(a,i,j) {
    val = a[i]
    h = i
    for k = i + 1 to j
        if (a[k] < val) {
            h = h + 1
            swap(a[h],a[k])
        }
    swap (a[i],a[h])
    return h
}
```

Algorithm 6.2.4 Quicksort

This algorithm sorts the array

$$a[i], \ldots, a[j]$$

by using the partition algorithm (Algorithm 6.2.2).

```
Input Parameters: a,i,j
Output Parameters: a
quicksort(a,i,j) {
    if (i < j) {
        p = partition(a,i,j)
        quicksort(a,i,p - 1)
        quicksort(a,p + 1,j)
    }
}
```

Example 7 - Quick Sort

h	k			
4	10	3	1	5

Val = 4

h		k		
4	10	3	1	5

Val = 4

	h	k		
4	3	10	1	5

Val = 4

	h		k	
4	3	10	1	5

Val = 4

		h	k	
4	3	1	10	5

Val = 4

		h		k
4	3	1	10	5

Val = 4

		h		k
1	3	4	10	5

Val = 4

Quick Sort

In the worst case, the partition element is placed at either the end or the start of the array, in either case, the first time partition is called, it takes $n - 1$ steps, the second time $n - 2$, the third $n - 3$ etc., giving:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$$

In reality, quicksort tends to perform much better than this.

Random Quicksort

We can make an "improvement" to quicksort by randomly choosing the partition value, rather than choosing the first value in the array.

Of course the worst case time is still $\mathrm{T}(n^2)$, but we can *expect* random quicksort to run in time $\mathrm{T}(n \log n)$.The difference is that now the running time is independent of the order of the input.

A Lower Bound for Sorting

For any *comparison* based sorting algorithm (this is not the only kind, just the most common and "useful") the running time is at least as large as the number of comparisons that must be done to sort.

If we consider the decision tree that represents the running of the algorithm (where each non leaf node is a comparison), then there must be at least h comparisons, where h is the height of the tree.

As there are n elements in the array, there must also be $n!$ possible arrangements, thus there are $n!$ leaves in the decision tree, corresponding to all sortings.

As a decision tree is a binary tree, we have that for a tree with t leaves and height h, $\lg t = h$.

A Lower Bound for Sorting

Thus we have $\lg(n!) \leq h$.

We have already shown that $\lg(n!) = \Theta(n \log n)$.

Thus for some c such that $c \times n \times \log n \leq \lg(n!)$.

Therefore $c \times n \times \log n \leq h$.

Thus the number of comparisons is lower bounded by $n \log n$.

So we can do no better than this for comparison based sorting.

Sorting Algorithms

Main algorithmic strategies used for sorting:

- Brute Force
  - Selection sort $O(n^2)$
  - Bubble sort $O(n^2)$
- Divide-and-Conquer
  - Mergesort $O(n \log n)$
  - Quicksort $O(n \log n)$, the worst case $O(n^2)$
- Decrease-and-Conquer
  - Insertion sort $O(n^2)$
- Transform-and-Conquer
  - Heapsort $O(n \log n)$

See animations at https://www.toptal.com/developers/sorting-algorithms

Which of these algorithms are stable?

Sorting Algorithms

- Straightforward algorithms are

  $\mathrm{O}(n^2)$

- More complex algorithms are

  $\mathrm{O}(n \log n)$

- Can we do better than that?

Counting and Radix Sort

Counting sort sorts an array of integers. These integers will be used as indices in an auxiliary array.

It first counts the number of occurrences of each integer value in the array, and then the number of values less than or equal to a given value.

Example 8

5	8	3	8	10	7

- The number of occurrences of each value k in the array:
  `c[3]=1, c[5]=1, c[7]=1, c[8]=2, c[10]=1`

- The number of elements less than or equal to k:
  `c[3]=1, c[5]=2, c[7]=3, c[8]=5, c[10]=6`

- Then we place element 5 in position 2, and decrement `c[5]`; we place 8 in position 5 and decrement `c[8]`; and so on.

Example 8

5	8	3	8	10	7	$c[3]=1$, $c[5]=2$, $c[7]=3$, $c[8]=5$, $c[10]=6$
	5					$c[3]=1$, $c[5]=1$, $c[7]=3$, $c[8]=5$, $c[10]=6$
	5			8		$c[3]=1$, $c[5]=1$, $c[7]=3$, $c[8]=4$, $c[10]=6$
3	5			8		$c[3]=0$, $c[5]=1$, $c[7]=3$, $c[8]=4$, $c[10]=6$
3	5		8	8		$c[3]=0$, $c[5]=1$, $c[7]=3$, $c[8]=3$, $c[10]=6$
3	5		8	8	10	$c[3]=0$, $c[5]=1$, $c[7]=3$, $c[8]=3$, $c[10]=5$
3	5	7	8	8	10	$c[3]=0$, $c[5]=1$, $c[7]=2$, $c[8]=3$, $c[10]=5$

Is this algorithm stable? If not what can we change so that it becomes stable?

Algorithm 6.4.2 Counting Sort

This algorithm sorts the array `a[1], ... , a[n]` of integers, each in the range `0` to `m`, inclusive.

```
Input Parameters: a,m
Output Parameters: a
counting_sort(a,m) {
        // set c[k] = the number of occurrences of value k
        // in the array a.
        // begin by initializing c to zero.
        for k = 0 to m
                c[k] = 0
        n = a.last
        for i = 1 to n
                c[a[i]] = c[a[i]] + 1
        // modify c so that c[k] = number of elements ≤ k
        for k = 1 to m
                c[k] = c[k] + c[k - 1]
        // sort a with the result in b
        for i = n downto 1 {
                b[c[a[i]]] = a[i]
                c[a[i]] = c[a[i]] - 1
        }
        // copy b back to a
        for i = 1 to n
                a[i] = b[i]
}
```

Counting Sort

- The complexity of counting sort is $\Theta\,(n+m)$, where n is the number of elements in the array each being in the range 0 to m.

- Is counting sort stable?

Algorithm 6.4.4 Radix Sort

This algorithm sorts the array $a[1], \ldots, a[n]$ of integers. Each integer has at most k digits.

```
Input Parameters: a,k
Output Parameters: a
radix_sort(a,k) {
    for i = 0 to k - 1
        counting_sort(a,10) // key is digit in 10^i's place
}
```

Radix Sort

- The complexity of radix sort is $\Theta(k \times n)$ where n is the number of integers and k is the max number of digits of the integers.

- Is Radix sort stable?

Selection

Random Select uses random partition to find the k^{th} smallest element in an array.

Algorithm 6.5.2 Random Select

Let *val* be the value in the array *a*[*i*], ... , *a*[*j*] that would be at *index k* ($i \le k \le j$) if the entire array was sorted. This algorithm rearranges the array so that *val* is at index *k*, all values at indexes less than *k* are less than *val*, and all values at indexes greater than *k* are greater than or equal to *val*. The algorithm uses the random-partition algorithm (Algorithm 6.2.6).

```
Input Parameters: a,i,j,k
Output Parameter: a
random_select(a,i,j,k) {
    if (i < j) {
        p = random_partition(a,i,j)
        if (k == p)
            return
        if (k < p)
            random_select(a,i,p - 1,k)
        else
            random_select(a,p + 1,j,k)
        }
}
```

Complexity of Random Select

Worst-case: $\Theta(n^2)$

Average: $\Theta(n)$