Search

goal: Find element in unsorted array

Linear Search (x, A)

```
for i in 0..< A.count
  if A[i] = x : return i
return false
```

Time complexity $O(n)$
"worst-case optimal"

goal: Find element in sorted array

Binary Search (x, A)

```
if A.count = 0 : return false or index in initial A
i ← ⌊A.count / 2⌋
if A[i] = x : return i
if x < A[i] : return BinarySearch (x, A[0... i-1])
return BinarySearch (x, A[i+1..< A.count])
```

Time complexity $O(\log n)$

Variant: If array elements are distributed evenly (i.e. linearly interpolatable), modify i to

$$i^* \leftarrow \frac{(x - A[0]) \cdot A.count}{A[A.count-1] - A[0]}$$

Time complexity $O(n)$ worst case, $O(\log\log n)$ with appropriate A

Sorting

Bubble Sort (A)

```
for j in 1..< A.count - 1
  for i in 0..< A.count - j
    if A[i] > A[i+1]
      Swap A[i] and A[i+1]
```

Selection Sort (A)

```
for i in 0..< A.count - 1
  j ← index of min(A[i..< A.count])
  Swap A[i] and A[j]
```

Insertion Sort (A)

```
for i in 1..< A.count
  j ← BinarySearch (A[i], A[0..i-1])
  x ← A[i]
  move A[j ... i-1] to A[j+1 ... i]
  A[j] ← x
```

Time complexity $O(n^2)$ for all above

HeapSort(A)

```
for i in ⌊(A.count − 1)/2⌋ ... 0
    RestoreHeapCondition(A, i, A.count − 1)
for i in A.count − 1 ... 1
    swap A[0] and A[i]
    RestoreHeapCondition(A, 0, i − 1)
return A
```

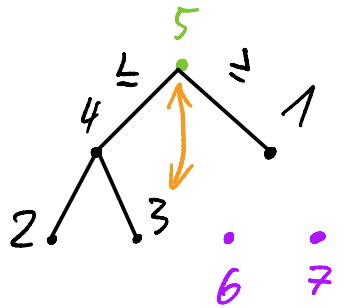
Time complexity $O(n \log n)$
(but worse locality!)

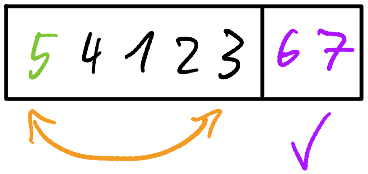
MergeSort(A)

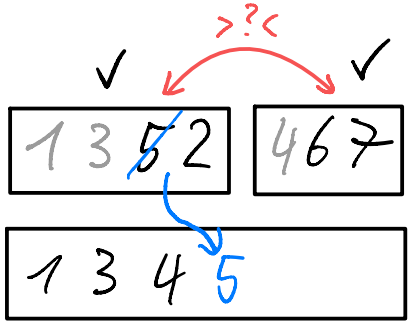
```
partitionSize ← 1
while partitionSize < A.count
    right ← −1
    while right + partitionSize < A.count
        left ← right + 1
        middle ← left + partitionSize − 1
        right ← min(middle + partitionSize, A.count − 1)
        Merge(A, left, middle, right)
    partitionSize ← partitionSize · 2
return A
```

RestoreHeapCondition(A, elementIndex, barrierIndex)

```
while 2 · elementIndex ≤ barrierIndex
    successorIndex ← 2 · elementIndex
    if successorIndex + 1 ≤ barrierIndex
    and A[successorIndex] < successorIndex + 1]
        successorIndex ← successorIndex + 1
    if A[elementIndex] ≥ A[successorIndex]
        return A
    Swap A[elementIndex] and A[successorIndex]
    elementIndex ← successorIndex
```







Merge(A, left, middle, right)

```
lp ← left; rp ← middle + 1; pos ← 0
while lp ≤ middle and rp ≤ right
    if A[lp] < A[rp]: B[pos] ← A[lp]; lp ← lp + 1
    else: B[pos] ← A[rp]; rp ← rp + 1
    pos ← pos + 1
while lp ≤ middle: B[pos] ← A[lp]; lp ← lp + 1; pos ← pos + 1
while rp ≤ right: B[pos] ← A[rp]; rp ← rp + 1; pos ← pos + 1
for i in left ... right: A[i] ← B[i − left]
```

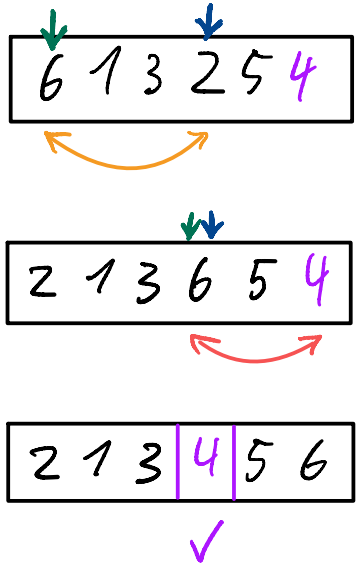
Time complexity $\Theta(n \log n)$
(but needs $\Theta(n)$ extra space)

QuickSort (A, l, r)

```
if l < r
    k ← Partition (A, l, r)
    Quicksort (A, l, k-1)
    Quicksort (A, k+1, r)
```

Partition (A, l, r)

```
i ← l
j ← r-1
pivot ← A[r]
repeat until break
    while i < r and A[i] < pivot :  i ← i+1
    while j > l and A[j] > pivot :  j ← j-1
    if i < j :  Swap A[i] and A[j]
    else :  break
Swap A[i] and A[r]
return i
```



Time Complexity $O(n^2)$

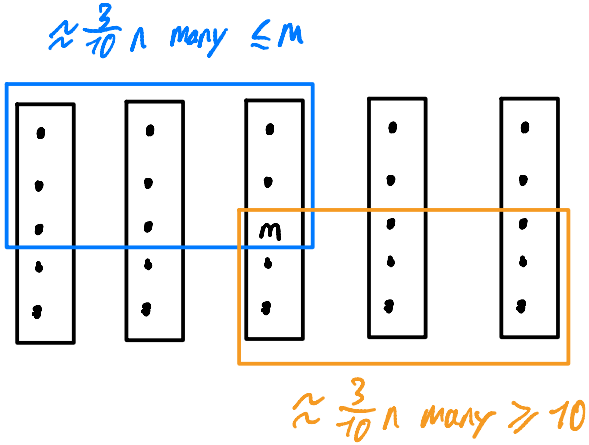
(if pivots are chosen at random usually $O(n \log n)$; uses $\Theta(n)$ extra space in this implementation)

Selection Problem

goal: Find k-th smallest / greatest element in an array.

Median Of Medians (A)

Split A in groups of $n \geq 5$ and determine each group's median
$A' \leftarrow$ all above group medians
if A'.count = 1 : return A'[0]
else: return MedianOfMedians



Time Complexity $O(n)$ worst and best-case
↳ hence not used IRL

Dynamic Programs

MinEditDist(word1, word2)

```
d[0][j] ← j ; d[i][0] ← i
for i in 1...word1.length
  for j in 1...word2.length
    replace ← d[i-1][j-1]
    if word1[i-1] ≠ word2[j-1]
      replace ← insert + 1
    delete ← d[i-1][j] + 1
    insert ← d[i][j-1] + 1
    d[i][j] ← min(replace, delete, insert)
return d[word1.length][word2.length]
```

	-	H	A	N
-	0	1	2	3
H	1	0	1	2
E	2	1	1	②

Note: LCS is essentially analogous to MED!

Time complexity $O(n^2)$

LongestCommonSubsequence(word1, word2)

```
d[i][0] ← 0 ; d[0][j] ← 0
for i in 1...word1.length
  for j in 1...word2.length
    d[i][j] ← max(d[i-1][j],
                  d[i][j-1],
                  d[i-1][j-1]
                    + 1 if word1[i-1] = word2[j-1])
return d[word1.length][word2.length]
```

LongestIncreasingSubseq(A)

```
endForLength[1] ← A[0]
head ← 1
maxLength ← 1
for el in A[1...]
  while head ≥ 1 and el ≤ endForLength[head]
    head ← head - 1
  head ← head + 1
  while head ≤ maxLength and el > endForLength[head]
    head ← head + 1
  endForLength[head] ← el
  maxLength ← max(maxLength, head)
return maxLength
```

0 1 0 2 3 ⇒

1	2	3	④	5
0	1	2	3	
0				

length ↓ overrides

Time complexity $O(n \log n)$

Subset Sum (k, A)

```
d[0][j] ← true  (for j ≥ 1)
d[i][0] ← true
for i in 1... A.count
  for j in 1...k
    num ← A[i]
    if j - num < 0 : canBeAdded ← false
    else: canBeAdded ← d[i-1][j-num]
    needntBeAdded ← d[i-1][j]
    d[i][j] = canBeAdded ∨ needntBeAdded
return d[A.count][k]
```

	0	1	2	3	4	5
–	1	0	0	0	0	0
2	1	0	1	0	0	0
1	1	1	1	1	0	0
3	1	1	1	1	1	(1)

Time complexity $O(kn)$

$O(2^n n)$ für $k = 2^n$

$O(n^{c+1})$ für $k = n^c$

Knapsack Maximum Value (n, W, V, k)

```
d[i][0] ← 0 ;  d[0][j] ← 0
for i in 1...n
  for j in 1...k
    maxValWithoutThis ← d[i-1][j]
    maxValWithThis ← d[i-1][j - W[i]] + V[i]
    d[i][j] = max (maxVal(withoutThis,
                   maxValWithThis)
return d[n][k]
```

Time complexity $O(nk)$

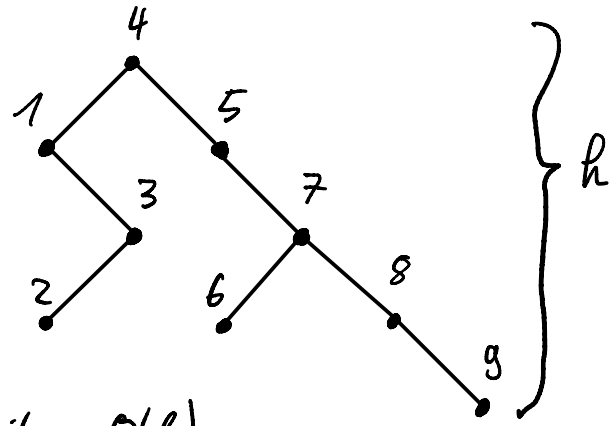
Knapsack FPTAS (ε, n, W, V, k)

```
v_max ← max(V)
m ← ε · v_max / n
for i in 1...n
  V'[i] ← ⌊V[i] / m⌋
return KnapsackMaxValue (n, W, V', k)
```

Time complexity $O(\frac{n^3}{\varepsilon})$

Binary Trees

goal: allow quick dictionary operations: search, insert, remove
↳ $O(\log n)$



Search Tree

condition: every node's descendants are $<$ on the left
$>$ on the right

Time complexity $O(h)$
↳ $O(n)$ worst-case

Search (key, root)

```
if root = null : return null
if root.key = key or null : return root
if x < root.key : return Search(key, root.left)
return Search(key, root.right)
```

Insert (key, root)

```
Leaf ← Search(key, root)
replace Leaf with new Leaf(key, null, null)
```

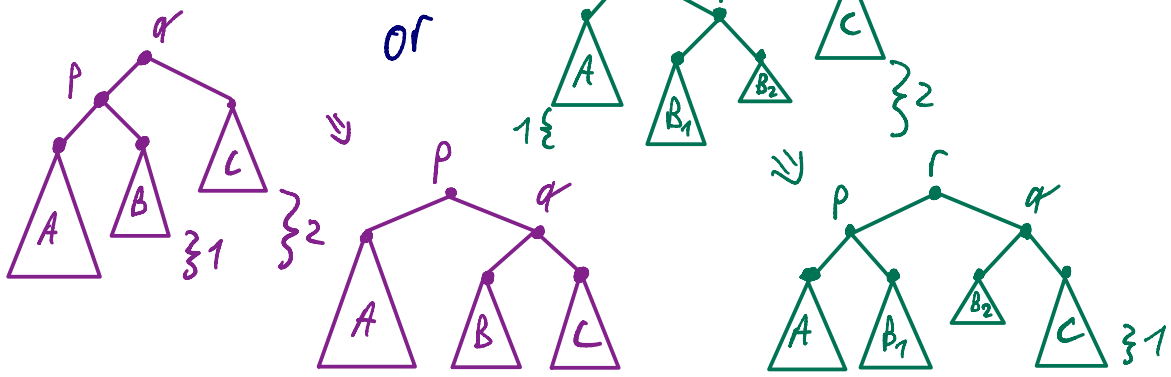
Remove (key, root)

```
node ← Search(key, root)
if node is a leaf
  remove the reference to node from node.parent
else if node has only one direct child
  replace the reference to node in node.parent
  with child
else
  repl ← node.right
  while repl.left ≠ null : repl ← repl.left
  node.key ← repl.key
  Remove(repl.key, root)
```

AVL-Tree

goal: keep tree semi-balanced, so everything runs in $O(\log n)$
condition: the height difference of every node's subtrees is in $[-1; 1]$

RestoreAVLConditionLeft()



Graph Traversal

```
BFS(G, s)
  Q ← {s}                          ◁ enqueue the first element!
  D ← {s}                          ◁ store previously discovered vertices!
  while Q ≠ ∅
    u ← dequeue(Q)                 (optionally store minimum distance)
    for (u,v) ∈ G.E                ◁ visit all new destinations
      if v ∉ D
        D ← D ∪ {v}                ◁ mark on discovery!
        enqueue(Q, v)              (optionally update minimum distance)
```

```
DFS(G, s)
  S ← {s}                          ◁ push the first element!
  V ← ∅                            ◁ store previously visited vertices!
  while S ≠ ∅
    u ← pop(S)
    if u ∉ V
      V ← V ∪ {u}                  ◁ mark on visit!
      for (u,v) ∈ G.E              (if order strictly matters, iterate in reverse)
        if v ∉ V
          push(S, v)
```

Example Implementation

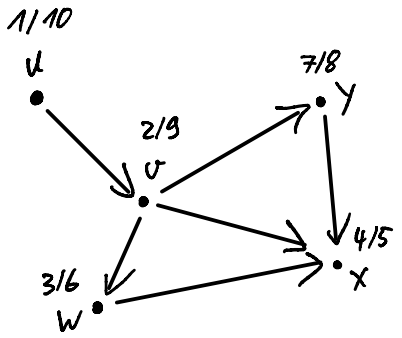
```
LinkedList<...> queue = ...;
queue.add(s);
boolean[] discovered = ...;
discovered[s] = true;
while (!queue.isEmpty()) {
  ... u = queue.removeFirst()
  for (... v : edges[u]) {
    if(!discovered[v]) {
      discovered[v] = true;
      queue.add[v];
    }
  }
}
```

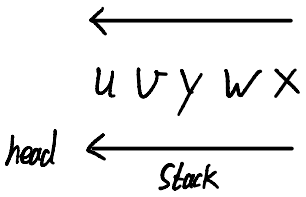
Time complexity $O(|V| + |E|)$

Directed Graph Utilities

Topological Sort (G)

```
R ← ∅                                  ◁ only exists on acyclic graphs!
V ∈ ∅                                  ◁ build the solution on a stack
for u ∈ G.V                            (not necessary if only one known source exists)
    if u ∉ V
        Recur(u)
            V ∈ V ∪ {u}
            for (u,v) ∈ G.E
                if v ∉ V
                    Recur(v)           ◁ use recursion to explore depth-first
            push(R,u)
return popAll(R)                       ◁ return in reverse post order
```





pre / post

(v,x) forward
(y,x) cross

Time complexity $O(|V|+|E|)$

Shortest Paths

	one-to-all	
$G=(V,E)$	BFS	$O(\|V\|+\|E\|)$
$G=(V,E,c)$ $c: E \to \mathbb{R}^+$	Dijkstra	$O((\|V\|+\|E\|)\log\|V\|)$ or $O(\|E\|+\|V\|\log\|V\|)$
$G=(V,E,c)$ $c: E \to \mathbb{R}$	Bellman-Ford	$O(\|V\|\|E\|)$

all-to-all	
repeated BFS	$O(\|V\|^2+\|V\|\|E\|)$
repeated Dijkstra	$O(\|V\|^2\log\|V\|+\|V\|\|E\|)$
repeated Bellman-Ford	$O(\|V\|^2\|E\|)$
Floyd-Warshall	$O(\|V\|^3)$
Johnson	$O(\|V\|^2\log\|V\|+\|V\|\|E\|)$

Dijkstra (G, s)

```
Q ← {(s, 0)}                          ◁ use a priority queue
D ← {s}                               ◁ mark vertices on discovery
d[s] ← 0                              ◁ store upper bounds for
for u ∈ G.V \ {s}                       minimum distances
  d[u] ← ∞
while Q ≠ ∅
  u ← extractMin(Q)                   ◁ grab the element of lowest
  for (u, v) ∈ G.E                      distance bound
    d* ← d[u] + G.w(u, v)
    if v ∉ D
      d[v] ← d*
      D ← D ∪ {v}
      enqueue(Q, (v, d*))             ◁ add v with its priority to
    else if d* < d[v]                   the queue
      d[v] ← d*
      decreaseKey(Q, (v, d*))         ◁ update v's priority in the
return d                                queue
```

Implementation Details

Java offers PriorityQueue<...>, but cannot decreaseKey(...). Workaround: Store references to most current Pair<...,...> in an array and call remove(...) and add(...). Otherwise the heap cannot be updated!

Alternatively, check on dequeue whether the vertex has already been dealt with (usually faster!)

With Fibonacci-Heap, Dijkstra may run in $O(|V| \log |V| + |E|)$.
With min-Heap we have $O((|V|+|E|) \log |V|)$

BellmanFord(G, s)

```
D ← {s}
d[s] ← 0
for u ∈ G.V \ {s}
  d[u] ← ∞
for _ in 1...|G.V|-1
  for (u,v) ∈ G.E
    d* ← d[u] + G.w(u,v)
    if d* < d[v]
      d[v] ← d*
      D ← D ∪ {v}
for (u,v) ∈ G.E
  d* ← d[u] + G.w(u,v)
  if d* < d[v]
    throw "Negative cycle detected"
return d
```

FloydWarshall(G)

```
for u ∈ G.V
  d[u][u] ← 0
for (u,v) ∈ G.E
  d[u][v] ← w(u,v)
for k in 0..<|G.V|
  for i in 0..<|G.V|
    for j in 0..<|G.V|
      d[i][j] ← min(d[i][j], d[i][k] + d[k][j])
return d
```

Note:

$d_{ij} := \min(d_{ij}, d_{ik} + d_{kj})$ shortest paths

$d_{ij} := d_{ij} \vee (d_{ik} \wedge d_{kj})$ a relation's transitive closure

$d_{ij} := d_{ij} + d_{ik} \cdot d_{kj}$ matrix multiply

Raising an adjacency matrix by the power of n reveals the number of paths of length n between vertices.

Johnson(G)

```
z ← new Vertex
for v ∈ G.V
  G.E ← G.E ∪ {(z,v)}
  G.w(z,v) ← 0
G.V ← G.V ∪ {z}
h ← BellmanFord(G, z)
for (u,v) ∈ G.E
  if u=z : G.E ← G.E \ {(u,v)}
  else : G.w(u,v) ← w(u,v) + h(u) - h(v)
G.V ← G.V \ {z}
for v ∈ G.V : d[v] ← Dijkstra(G, v) - h(u) + h(v)
return d
```

Minimum Spanning Trees

Boruvka (G) (non-heap version)

```
F ← ∅ ; S ← ∅                    ◁ Keep track of all components
for u ∈ G.V
  S ← S ∪ {u}
while S.count > 1
  for s* ∈ S                     ◁ Address every component
    (u*, v*) ← null ; w* ← ∞
    for (u,v) ∈ G.E \ {F}
      if repr(S,u) = s* ≠ repr(S,v) and w(u,v) < w*
        (u*, v*) ← (u,v) ; w* ← w(u,v)
    F ← F ∪ {(u*, v*)}
    union(S, s*, repr(S,v))
return F
```

Kruskal (G)

Time complexity $O(|V||E| + |E|\log|E|)$

```
F ← ∅ ; S ← ∅
for u ∈ G.V
  S ← S ∪ {u}
E' ← sort Ascending (G.E, G.w)   ◁ Focus on edge weights
for (u,v) ∈ E'
  if repr(S,u) ≠ repr(S,v) : F ← F ∪ {(u,v)}
return F
```

Prim (G, s) (Dijkstra-esque version)

```
F ← ∅                            Use a priority queue to quickly
Q ← heapify(G.V, ∞)              ◁ extract minimum edges to the component from outside
decreaseKey(Q, (s, 0))           ◁ Keep track of one component
d[s] ← 0
for u ∈ G.V \ {s}
  d[u] ← ∞
  p[u] ← null                    ◁ Store each safe vertex' predecessor
while Q ≠ ∅
  u ← extractMin(Q)
  for (u, v) ∈ G.E
    d* ← G.w(u,v)
    if p[v] = null and d* < d[v]
      d[v] ← d*
      p[v] ← u
      decreaseKey(Q, (v, d*))
for u ∈ G.V \ {s}
  F ← F ∪ {(p[u], u)}
return F
```

Time complexity $O((|V|+|E|)\log|V|)$ (Boruvka and Prim)