Algorithms and Data Structures

Daniel Balle 2018

Heap

A heap is a nearly complete tree in which all parents have values either larger (*max-heap*) or lower (*min-heap*) than their children.

heapify	**build**	**heapsort**	**search**	**modifiy**
$\log n$	n	$n \log n$	n	$\log n$

Remark. Heaps can be implemented efficiently using *arrays*.

Space Efficient Trees

Nearly complete binary trees can be implemented efficiently using an array and the following helper functions.

```
children(i) = { 2i + 1, 2i + 2 }
parent(i)   = ⌊(i-1) / 2)⌋
```

Remark. code for 0 indexes arrays.

Heapify

The `heapify` function produces a new *heap* given an arbitrary root to two valid heaps in $O(\log n)$ by iteratively swapping the root with its largest child.

```
node = largest(root, left(root), right(root))
if (root != node) {
    exchange heap[root] and heap[node]
    heapify(node)
}
```

Remark. This function is the core of `heap-build` and `heapsort`.

Building a Heap

To build a *heap* in linear time $O(n)$ we iteratively apply `heapify` from the parents of leafs, which are valid heaps, to the root.

```
for i from parent(n) to 1
   heapify(i)
```

Heapsort

average	worst	memory	stable
$n \log n$	$n \log n$	1	$\times$

`heapsort` is a sorting algorithm using a *heap* to iteratively extract the root and rebuilding a smaller heap using `heapify`.

```
build-max-heap(A)
heap-end = n - 1
while (end > 0) {
   swap A[heap-end] and A[0]
   heap-end--
   heapify(A, heap-end)  // restore heap property
}
```

Selection Sort

average	worst	memory	stable
n^2	n^2	1	✓

`selection-sort` is an inefficient sorting algorithm progressively finding the *smallest* element to grow a sorted subarray.

```
for (i from 0 to n-2)
    x = a[i]
    for (j from i to n-1) x = min(x, a[j])
    a[i] = x
```

Remark. Similar to *insertion sort*. A bidirectional variant finding both minimum and maximum each iteration is *cocktail sort*.

Quicksort

average	worst	memory	stable
$n \log n$	n^2	1	$\times$

`quicksort` is a sorting algorithm progressively partitioning the array into two subarrays containing only elements respectively smaller and larger than some pivot.

```
quicksort(A, lo, hi):          // if lo < hi
    p = partition(A, lo, hi)   // pivot at correct spot
    quicksort(A, lo, p-1)
    quicksort(A, p+1, hi)
```

Linear partitioning schemes such as *Lomuto* and *Hoare* produce an average running time $T(n) = O(n) + 2T(n/2)$.

Remark. Randomized quicksort also yields $O(n \log n)$ worst-case.

Lomuto Partitioning

Lomuto is a linear partitioning scheme using the last element as the pivot and progressively growing a region with only lower elements.

```
 p = A[hi]
 i = lo  // A[lo..i-1] are elements below p
 for (j from lo to hi - 1)  // A[i..j] are over p
     if (A[j] < p)
         exchange A[i] and A[j]
         i += 1
exchange A[i+1] and p
return i+1
```

Remark. Used in *quicksort* and *quickselect*. By fist swapping a random element to the end we produce randomized quicksort.

Remark. Less efficient than *Hoare*.

Hoare Partitioning

Hoare is a *linear* partitioning scheme in which two pointers travel towards each other while exchanging elements that violate their respective relation to the pivot.

```
p = A[lo]
i = lo - 1  // A[lo..i] are smaller than p
j = hi + 1 // A[j..hi] are larger than p
while True
    do i++ while A[i] < p
    do j-- while p < A[j]
    if (i < j) exchange A[i] and A[j]
```

Remark. Used for *quicksort* and *quickselect*.

Dutch Flag Partitioning

The Dutch Flag problem is solved by a *linear* three-way partition operating with constant memory which iterates over the array while progressively growing three regions.

```
x = -1  // A[0..x] contains 0s
i = 0   // A[x+1...i-1] contains 1s
y = n   // A[y..n] contains 2s
while (i < y)
    if (A[i] < 1) { x++; swap A[x] and A[i]; i++ }
    if (A[i] = 1) { i++ }
    if (A[i] > 1) { y--; swap A[y] and A[i] }
```

Remark. Useful for *quicksort* with multiple duplicates.

Quickselect

Quickselect or Hoare Selection uses a partitioning scheme such as *Lomuto* or *Hoare* to select the k-th element in linear $O(n)$ time.

```
select(A, lo, hi, k):
    if (lo == hi) return A[lo]
    p = partition(A, lo, hi)  // pivot at correct spot
    if (p == k) return A[p]
    else if (p < k) return select(A, p+1, hi, k)
    else return select(A, lo, p-1, k)
```

A pivot selection strategy such as *median-of-medians* can be used.

Remark. Worst case $O(n^2)$ as for *quicksort*. Constant memory overhead under tail call optimization or iteration.

Remark. To find the k-th element we can also use a *heap* of size k.

Mergesort

average	worst	memory	stable
$n \log n$	$n \log n$	n	✓

Mergesort is a stable sorting algorithm recursively sorting two sub-arrays of equal size before merging them.

```
merge-sort(A, lo, hi):
    q = ⌊(lo+hi)/2⌋
    merge-sort(A, lo, q)
    merge-sort(A, q+1, hi)
    merge(A, lo, hi, q)
```

Linear merging can be performed with $O(n)$ memory and sentinel cards. `merge-sort` thus has running time $T(n) = 2T(n/2) + O(n)$.

Remark. Practical for multi-threaded sorting.

Counting Sort

running time	memory	stable
$n + k$	$n + k$	✓

Counting sort is a stable integer sorting algorithm for a known range $[0, k]$ placing elements based on their prefix sum.

```
for (i from 1 to A.length) C[A[i]] += 1 // occurences
for (j from 1 to k) C[j] += C[j-1] // prefix sum
for (i from A.length to 1)
    B[C[A[i]]] = A[i]  // put A[i] at position C[A[i]]
    C[A[i]] -= 1
```

Remark. Using both 0 and 1 indexed array. Due to being stable, counting sort can be used for radix sort.

Remark. *Prefix* sums can be very useful in subarray problems.

Radix Sort

running time	memory	stable
$d(n+k)$	$n+k$	✓

Radix sort is an integer sorting algorithm using *counting sort* to sort elements for every digit, beginning with the *least-significant*.

```
for (i from 1 to d)
    stable-digit-sort(A, i) // digit 1 is the LSD
```

Remark. The underlying sorting algorithm needs to be stable.

Bucket Sort

Bucket sort is a sorting algorithm assuming the input is drawn from a uniform distribution over $[0, 1)$. Distribute the input numbers into n equal-sized subintervals and sort each.

```
for (i in 1 to n) insert A[i] into list B[⌊n A[i]⌋]
for (i in 1 to n) sort list B[i]
return B[0], ..., B[n-1]
```

Remark. Each bucket i is expected to contain few elements n_i, yielding linear $O(n)$ average running time.

$$E\left[T(n)\right] = \Theta(n) + \sum_{i}^{n-1} O(E[n_i^2])$$

Breadth-First Search

running time $O(V + E)$ or $O(b^{d+1})$

`BFS` is a graph traversal and search algorithm progressively expanding the *shallowest* nodes using a FIFO queue for the frontier.

```
frontier = queue(s)
discovered = {s : s}          // distance optional
while (frontier):
   u = frontier.dequeue()
   for (v in u.neighbors if v not in discovered)
       frontier.append(v)
       discovered[v] = u
       if (v == t) return t  // optional
```

Remark. Produces a *breadth-first tree*. BFS can find the *shortest path* if path cost is a non-decreasing function of depth.

Depth-First Search

running time $O(V+E)$

`DFS` is a graph traversal and search algorithm progressively expanding the *deepest* nodes in the frontier.

```
dfs-visit(u):
    u.discovery = ++time
    for (v in u.neighbors if v.parent = ∅)
         v.parent = u
         dfs-visit(v)
    u.finish = ++time
for (u in G.V if v.parent = ∅) dfs-visit(u)
```

Remark. A non-recursive implementation uses a stack.

Remark. Applications include *topological sort*, finding strongly connected *components* or labeling *node sets, i.e. deapth-first trees*.

Topological Sort

`topological-sort` produces a linear ordering $\prec$ in a directed acyclic graph G such that $(u, v) \in E \implies u \prec v$, using decreasing finishing times of a *depth-first* forest.

```
topological-visit(u):
   u.discovered = true
   for (v in u.neighbors if not v.discovered)
       topological-visit(v)
   ordering.prepend(u)
```

Strongly Connected Components

Depth-first search can be used to identify strongly connected components by being applied to the transposed graph $G^\top$.

```
dfs(G) to produce u.f
dfs(G^T) but create each tree by decreasing u.f
each tree is a strongly connected component
```

Remark. $G^\top$ simply contains all edges of G reversed.

Graphs in AI

Refer to the **Artificial Intelligence** ↑ notes for more details on both uninformed and *informed* search algorithms.

Iterative Postorder Traversal

To implement an *iterative* postorder traversal of a binary tree we can use a stack to perform *depth-first search* and order by decreasing discovery time.

```
stack = [root]
while (stack)
    x = stack.pop()
    solution.prepend(x)
    if (x.left) stack.push(x.left)
    if (x.right) stack.push(x.right)
```

Remark. Preorder traversal is achieved using the same logic. Consider the similarities between recursion and a stack here.

Disjoint-Set Data Structure

A *disjoint-set* or *union-find* data structure tracks a set of elements partitioned into a number of disjoint subsets using a *representative* node for every subset.

make set	**find**	**union**
1	n^*	n^*

(*) if optimized $\alpha(n)$

Remark. Used by *Kruskal's algorithm* for minimum spanning trees.

Data Structure, Set, Disjoint-Set, Union-Find

Disjoint-Set Structure Find

The `find` function of a *disjoint-set* data structure determines the representative of the set for a given element `x`.

```
find(x):
   if (x.parent != x) x.parent = find(x.parent)
   return x.parent
```

Remark. This `find` uses *path compression* to flatten the tree structure. Alternate optimizations are *path halving* and *path splitting*.

Data Structure, Set, Disjoint-Set, Union-Find

Disjoint-Set Structure Union

The `union` function of a *disjoint-set* data structure merges two sets by determining a common representative by *rank* or *size*.

```
union(x, y)  // by size
    xroot, yroot = find(x), find(y)
    if (xroot.size < yroot.size) swap(xroot, yroot)
    yroot.parent = xroot
    xroot += yroot.size
```

Data Structure, Set, Disjoint-Set, Union-Find

Kruskal’s Algorithm

running time $O(E \log E)$

Kruskal’s algorithm finds the *minimum spanning tree* of a graph by greedily adding edges of increasing weight using *Union-Find*.

```
for v in V: make-set(v)
for (u,v) in E ordered by weight:
    if find(u) != find(v):
         A = A ∪ {(u,v)}
         union(u, v)
```

Remark. See related *Prim’s algorithm*.

Prim's Algorithm

running time $O(E \log V)$ or $O(E + V \log V)$

Prim's algorithm finds the *minimum spanning tree* of a graph by starting at any vertex and greedily adding the cheapest *connection*.

```
C[v] = $\infty$ for all v // cheapest connection to v
E[v] = $\emptyset$ for all v  // corresponding edge
Q = V
while Q:
    extract v from Q with min C[v]
    add v to F, and also E[v] if not $\emptyset$
    update E[w], C[w] for all (v, w) in E
```

Remark. See related *Kruskal's algorithm*.

Bellman-Ford Algorithm

running time $\Theta(VE)$

`bellman-ford` solves single-source shortest-path problems for any directed graph through relaxation, and detects negative cycles.

```
u.d = ∞ for (u in V) but s.d = 0
for (i from 1 to |V|-1)
    for (edge (u, v) in E)
        relax(u, v)  // can v.d be improved through u?

for (edge (u, v) in E)
    if v.d > u.d + w(u, v) raise CycleException
```

Remark. Relaxation is used greedily in *Dijkstra's* algorithm.

Dijkstra's Algorithm

running time $O(V^2)$ **or** $O(E + V \log V)$

`dijkstra` solves single-source shortest-path problems for directed graphs with non-negative weights using *greedy* relaxation.

```
u.d = $\infty$ for (u in V) but s.d = 0
S = $\emptyset$ // set of finished vertices
while (V-S)
    u = extract-min(V-S)  // min u.d in V-S
    if (u == t) return t  // optional
    S = S $\cup$ { u }
    for (v in u.neighbors) relax(u, v)
```

Remark. Similar to *BFS* and *uniform cost search* from **AI** ↑ notes.

Remark. The fastest running time is achieved using a min-priority queue with a Fibonacci heap.

Graphs, Searching, Shortest Path, Single-source, Relaxation, Greedy

DAG Shortest Path

running time $O(V + E)$

The single-source shortest-path problem for directed acyclic graphs can be solved linearly using *topological sort*.

```
u.d = ∞ for (u in V) but s.d = 0
for (u in topological-sort(G))
    for (v in u.neighbors)
        relax(u, v)  // can v.d be improved through u?
```

Floyd-Warshall Algorithm

running time $O(V^3)$

`floyd-warshall` solves all-pairs shortest-path problems for graphs without negative weight cycles using dynamic programming. Define d_{ij}^k as the shortest path from i to j using vertices $\{1, ..., k\}$.

$$d_{ij}^k = \min\left\{w_{ij}, d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\right\}$$

Hierholzer Algorithm

Given a directed graph, *Hierholzer's* algorithm finds an *Euler circuit*, i.e. a path that traverses every edge of the graph and ends on the starting vertex, in linear $O(E)$ time.

```
TODO!
```

Graphs, Traversal, Single-Source, Euler Circuit

Huffman Code

A Huffman code is an optimal *prefix* scheme for a character coding problem. The corresponding full binary tree is constructed by *greedily* merging leaves with minimal frequency.

```
Q = C  // the characters
for (i from 1 to n-1)
   create new node z
   z.left = x = Q.extract-min()  // 0
   z.right = y = Q.extract-min() // 1
   z.freq = x.freq + y.freq
   Q.insert(z, z.freq)
return Q.extract-min()  // root
```

Rod Cutting Problem

Given prices p_i for a rod of length i, determine the highest revenue decomposition for a rod of length n.

```
input:    p = [1, 5, 8, 9], n = 4
solution: r = 10  // two pieces of length 2
```

Solution ↓ .

Rod Cutting Solution

The *rod-cutting* problem can be solved using dynamic programming by exploiting optimal substructure. The maximum price r_n for a rod of length n can be written as:

$$r_n = \max_{1 \leq i \leq n} \{p_i + r_{n-i}\}$$

A running time of $\Theta(n^2)$ is achieved by both the *memoized top-down* and *bottom-up* implementations.

Memoized Rod Cutting

The top-down implementation `rod-cut` with memoization to the *rod-cutting* problem is written recursively but solutions to subproblems are remembered in `r[i]`.

```
if (r[n] $\geq$ 0) return r[n]
q = 0
for (i from 1 to n)
    q = max{q, p[i] + rod-cut(n - i)}
r[n] = q
return r[n]
```

Bottom-Up Rod Cutting

The bottom-up implementation `rod-cut` to the *rod-cutting* problem iteratively solves subproblems of larger size.

```
for (i from 1 to n)
    q = $-\infty$
    for (j from 1 to i)
         q = max(q, p[j] + r[i - j])
    r[i] = q
return r[n]
```

Longest Common Subsequence

Given two sequences $X = \langle x_1, ..., x_n \rangle$ and $Y = \langle y_1, ..., y_m \rangle$, determine the maximum-length common subsequence Z of X and Y.

```
input:    X = ABCBDAB, Y = BDCABA
solution: BCBA or BDAB  // 4
```

Solution ↓ .

LCS Solution

The *longest common subsequence* problem can be solved using dynamic problem in $\Theta(mn)$. Define c_{ij} the length of the LCS of the prefix sequences $X_{1..i}$ and $Y_{1..j}$.

$$c_{ij} = \begin{cases} c_{i-1,j-1} + 1 & \text{if } x_i = y_j \\ \max\{c_{i,j-1}, c_{i-1,j}\} & \text{if } x_i \neq y_j \end{cases}$$

For the optimal solution use auxiliary table $b_{ij} \in \{\uparrow, \nwarrow, \leftarrow\}$ to record the optimal structure of c_{ij} and backtrack from b_{nm}.

Edit Distance

Given two strings `x`, `y`, determine their *edit distance*, i.e. the minimum number of character operations $\in$ { `insert`, `replace`, `delete` } to transform one string into the other.

```
input: horse, ros
output: 3  // horse -> rorse -> rose -> ros
```

Solution ↓ .

Edit Distance Algorithm

We determine the *edit distance* between two strings using dynamic programming through `dp[i,j]` defined as the edit distance between the substrings `x[..i-1]` and `y[..j-1]`.

$$d_{i,j} = \begin{cases} d_{i-1,j-1} & \text{if } x_{i-1} = y_{j-1} \\ 1 + \min\{d_{i-1,j-1}, d_{i,j-1}, d_{i-1,j}\} & \text{otherwise} \end{cases}$$

Remark. Remember DP paradigm of using *right-bound* substrings!

Distinct Subsequences

Given two strings `s` and `t`, determine the number of distinct subsequences of `t` in `s`.

```
input: s = "babgbag", t = "bag"
solution: 5  // 124, 127, 167, 367, 567
```

Solution ↓ .

Distinct Subsequences DP

The number of *distinct subsequences* of `t` in `s` is determined through dynamic programming using `dp[i,j]` as the number of distinct subsequences of `t[..j-1]` in `s[..i-1]`.

$$d_{i,j} = \begin{cases} d_{i-1,j} & \text{if } s_{i-1} \neq s_{j-1} \\ d_{i-1,j} + d_{i-1,j-1} & \text{otherwise} \end{cases}$$

Remark. Initialize with `dp[i,j] = (j == 0) ? 1 : 0`.

Maximum Subarray Problem

Given a one-dimensional array of numbers, determine the continuous subarray with maximum sum.

```
input: [-2, 1, -3, 4, -1, 2, 1, -5, 4]
solution: 6  // [4, -1, 2, 1]
```

Solution ↓ .

Kadane's Algorithm

Kadane's algorithm is a linear $O(n)$ dynamic programming solution to the *maximum subarray* problem which iteratively determines the maximum subarray B_i *ending* at position i using B_{i-1}.

```
bi = s = A[0]
for (i from 1 to n-1)
   bi = max(bi + A[i], A[i])
   s = max(s, bi)
return s
```

Remark. `kadane` is an instance of a *sliding-window* algorithm.

Longest Increasing Subsequence

Given a one-dimensional array of numbers, determine the longest increasing subsequence.

```
input: [10, 22, 9, 33, 21, 50, 41, 60, 80]
solution: 6  // [10, 22, 33, 50, 60, 80]
```

Solution ↓ .

LIS Solution

The *longest increasing subsequence* problem can be solved using dynamic programming. Define x_i to be the smallest number terminating a LIS of length i.

```
x[1] = A[0];  l = 1             // length of best LIS
for (n in A)
    if (x[l] < n) x[++l] = n    // new LIS
    else
         i = candidate-lis(n)   // x[i-1] < n < x[i]
         x[i] = n               // improve that sequence
return x[l]
```

`candidate-lis` determines the longest LIS whose last element can be replaced by `n`. Since `x` is *sorted*, we use binary search.

Remark. An alternate solution uses `LIS[i]` as the length of the LIS ending at position `i` for a running time $O(n^2)$.

Longest k-Sum Subarray

Given a one-dimensional array of numbers `A`, determine the longest subarray `A[i..j]` summing to k.

```
input: A = [3, -5, 8, -14, 2, 4, 12], k = -5
solution: 5  // [-5, 8, -14, 2, 4]
```

Solution ↓ .

Longest k-Sum Subarray Solution

The *longest k-sum subarray* problem is solved in *linear* time by defining `P[s]` as the smallest index i such that subarray `A[0..i]` has prefix sum s.

```
prefix = best = 0
P = {0 : -1}
for (i from 0 to n-1)
    prefix += A[i]
    missing = prefix - k
    if (prefix not in P) P[prefix] = i
    if (missing in P) best = max(best, i - P[missing])
return best
```

Remark. Subarray `A[p..i]` has sum k where `p = P[missing]`.

Longest Substring Problem

Given a string `S`, determine the longest *or shortest* substring `S[i..j]` satisfying some criterion C.

```
input: S = ababbcabaac
criterion: pangram D = {a, b, c}
solution: cab  // 3
```

Example. Shortest pangram, longest substring without duplicates.

Solution ↓ .

Sliding Window Method

The general *longest substring* problem can be solved in linear time using a *sliding window* technique, where two pointers extend or retract some substring `S[i..j]` to satisfy C.

```
for (j from 0 to n-1)       // shortest pangram
    while (S[i..j] satisfies C)
         is S[i..j] new best? ; i++
```

```
for (j from 0 to n-1)       // longest no duplicates
    while (S[i..j] does not satisfy C) i++
    is S[i..j] new best?
```

Remark. Implementation depends on C and length optimization. Similar to *Kadane's* algorithm.

0-1 Knapsack Problem

Given sizes s_i and values v_i for n items, determine the maximum value a knapsack with capacity k can carry.

```
input: s, v = [(10, 60), (20, 100), (30, 120)]
capacity: k = 50
solution: 220  // item 2 and 3
```

Solution ↓ .

0-1 Knapsack Algorithm

The *knapsack* problem can be solved using dynamic programming. Define $d_{i,k}$ to be the optimal value for a knapsack of capacity k using only items $\{1, .., i\}$.

$$d_{i,k} = \max\{v_i + d_{i-1,k-s_i}, d_{i-1,k}\}$$

```
for (j from 1 to k)
    for (i from 1 to n)
        d[i, j] = ...  // also check edge-cases
```

Remark. Additional substructure dimension than the *rod-cutting* solution as each items can only be taken once.

Fractional Knapsack Problem

Given sizes s_i and values v_i for n items, determine the maximum value a knapsack with capacity k can carry if a *fraction* of each item can be taken.

```
input: s, v = [(10, 60), (20, 100), (30, 120)]
capacity: k = 50
solution: 240  // item 1, 2 and 2/3 of item 3
```

Solution ↓ .

Greedy Knapsack Algorithm

The *fractional knapsack* problem can be solved using a *greedy* strategy. Order items by their volumetric value density $x_i = v_i/s_i$ and greedily pick as much as possible of each.

```
sort items by v[i]/s[i]
i = 0;
while (capacity > 0)
   fraction = min(1, capacity/s[i])
   capacity -= fraction * s[i]
   i++
```

Interval Selection Problem

Given starting times s_i and finishing times f_i of n intervals, determine the maximum subset of mutually compatible intervals.

```
input: s = [1, 3, 0, 5, 8, 5]
       f = [2, 4, 6, 7, 9, 9]
solution: {0, 1, 3, 4}
```

Solution ↓ **.**

Interval Selection Algorithm

The *interval selection* problem can be solved by *greedily* picking compatible intervals by increasing finishing time f_i.

```
sort intervals by finishing time $f_i$
S = {A[0]}
f = f[0]
for (i from 1 to n)
    if (s[i] $\geq$ f)
        S = S $\cup$ {A[i]}
        f = f[i]
return S
```

Interval Partitioning Problem

Given starting times s_i and finishing times f_i of n intervals, determine the smallest partition into compatible intervals.

```
input: s = [1, 3, 0, 5, 8, 5]
       f = [2, 4, 6, 7, 9, 9]
solution: 3  // {2, 4} + {0, 1, 3} + {5}
```

Solution ↓ .

Interval Partitioning Algorithm

The *interval partition* problem can be solved by *greedily* assigning intervals to the first compatible set by increasing start time s_i.

```
sort intervals by starting time $s_i$
r = []  // when resources become available
for (i from 0 to n-1)
    for (j=0; j < r.length; j++)  // find compatible set
        if (r[j] $\leq$ s[i]) break
    assign i to j
    r[j] = f[i]
```

Remark. See related problem *Meeting Rooms*.

Meeting Rooms

Given starting times s_i and finishing times f_i of n intervals, determine the maximum number of incompatible intervals.

```
input: s = [1, 3, 0, 5, 8, 5]
       f = [2, 4, 6, 7, 9, 9]
solution: 3  // (0, 6) × (5, 7) × (5, 9)
```

Solution ↓ .

Meeting Rooms Algorithm

To determine the maximum number of *incompatible intervals*, we *encode* each timestamp $t \in \{s_i, f_i\}$ to a pair `(t, {-1, 1})`.

```
bounds = []
for (interval in intervals)
    bounds.push((interval.s, 1))  // start => new room
    bounds.push((interval.f, -1)) // end => free room
sort(bounds)
rooms = 0, max_rooms = 0
for (bound in bounds)
    rooms += bound.second
    max_rooms = max(max_rooms, rooms)
```

Remark. See related problem *Interval Partitioning Problem*.

Non-Overlapping Intervals

Given starting times s_i and finishing times f_i of n intervals, determine the minimum number of intervals to be discarded to render all remaining non-overlapping.

```
input: [[1,2], [2,3], [3,4], [1,3]]
solution: 1 // remove [1, 3]
```

Solution ↓ .

Non-Overlapping Intervals Algorithm

To determine the minimum number of *intervals to be discarded* we can directly apply the solution to the *interval selection problem*, but instead return the opposite.

```
sort intervals by increasing ending time $f_i$
removed = 0, current_end = $-\infty$
for (i in intervals):
    if (i.start < current_end) removed++
    else current_end = max(current_end, i.end)
```

Next Greater Element

Given a one-dimensional array, determine for each integer the next greater element, i.e. the first larger element on its right.

```
input:    [5, 7, 4, 3, 6,  9, 2,  8]
solution: [7, 9, 6, 6, 9, -1, 8, -1]
```

Solution ↓ .

NGE Solution

The *next greater element* problem can be solved in linear time using a *stack* containing *pending* integers. As we traverse the array we compare the current element to the top unassigned elements.

```
 s = []  // contains (value, position)
 for (i from 0 to n-1)
     while (s && s.top()[0] < A[i])  // NGE for s.top
         x = s.pop()
         A[x[1]] = A[i]
     s.push(A[i], i)
while (s) A[s.pop()[1]] = -1
```

Longest Balanced Subarray

Given a binary array $\in \{0, 1\}^n$, determine the longest continuous subarray containing an equal number of each digit.

```
input: [0, 1, 0, 0, 1, 1, 0, 0]
solution: [1, 0, 0, 1, 1, 0]  // 6
```

Solution ↓ **.**

Balanced Subarray Solution

To solve the *longest balanced subarray* problem we use the algorithm to find the *longest 0-sum subarray* after substituting 0's by -1's.

```
B = {0: -1}
balance = longest = 0
for (i from 0 to n-1)
    balance += 1 if A[i] == 1 else -1
    if (balance in B)
         longest = max(longest, i - B[balance])
    else B[balance] = i
```

Remark. Note the use of an index map and prefix sums.

Inorder Successors Sum

Given the `root` of a binary tree, add to each `node` the sum of all its in-order successors.

```
input: inorder = [5, 7, 2, 9, 10, 3]  // given as tree
solution: [36, 31, 24, 22, 13, 3]
```

Solutions ↓ ↓ .

Recursive Inorder Successor Sum

The *in-order successor sum* problem can be solved *recursively* using an accumulator which each node increases by its own value.

```
int visit(node, acc):  // acc = sum of upper successors
    if (node == NULL) return acc
    acc = visit(node->right, acc)
    node->value += acc
    return visit(node->left, node->value)
```

Remark. `acc` acts as a global variable. Use a pointer in `C++`.

Remark. See the *iterative* solution.

Iterative Inorder Successor Sum

The *in-order successor sum* problem can be solved *iteratively* using a stack to traverse the tree in-reverse-order and an accumulator which each node increases by its own value.

```
digg(node, stack):
    while (node) {stack.push(node); node = node->right}

solve(root):
    acc = 0; stack = []; digg(root, stack)
    while (stack)
        node = stack.pop()
        node->value = acc = acc + node->value
        digg(node->left, stack)
```

Remark. See the *recursive* solution.

Tree, Traversal, Recursive, Solution

Longest Valid Parentheses

Given a string `s` containing characters `(` and `)`, determine the longest valid well-formed parenthesis substring.

```
input: S = "()(()()(()"
solution:     "()()"
```

Solution ↓ .

Longest Valid Parentheses Solution

The *longest valid parentheses* problem can be solved using an advanced *sliding window* method, using a stack containing the latest index of a potentially problematic character.

```
stack = [-1]  // sentinel
solution = -1
for (i from 0 to n-1)
    if (s[i] == "(") stack.push(i)
    else
        stack.pop()    // one less problem
        if (stack.empty()) stack.push(i) // problem!
        else solution = max(solution, i - stack.top())
```

k-Sum Combinations

Given a set of integers `S` and an integer `k`, find all distinct combinations from integers in `S` whose sum is equal to `k`.

Remark. A combination may contain duplicates.

```
input: S = [2, 3, 6, 7], k = 7
solution : [[7], [2, 2, 3]]
```

Solution ↓ .

k-Sum Combinations Solution

The *k-sum combinations* problem can be solved recursively using the helper function `solve`. Duplicates can be avoided by progressively restricting integers used from `S`.

```
solution = [], current = []
solve(k, j)
   if (k == 0) { solution.push_copy(current); return }
   for (i from j to n-1 if S[i] <= k)
       current.push_back(S[i])
       solve(k - S[i], i)   // avoid duplicates with j
       current.pop_back()   // clean-up
```

Remark. Note how only a single helper buffer `current` is needed.

Frog Problem

You are positioned at the beginning on an array `A` of non-negative integers, representing the maximum distance `A[i]` you can jump forward from each position `i`. Determine the minimum number of jumps to reach the last position.

```
input: [2, 3, 1, 1, 4]
solution: 2  // 0 -> 1 -> 4
```

Solutions ↓ ↓ ↓ ↓ .

Dynamic Frog Programming

The *frog problem* can be solved in $O(n^2)$ using dynamic programming with `X[i]` as the minimum jumps required from index `i`.

```
X[..] = ∞, X[n-1] = 0
for (i from n - 2 to 0)
    for (j from 1 to A[i] if i + j < n)
        X[i] = min(X[i], 1 + X[i+j])
return X[0]
```

Remark. More efficient solutions are also *provided*.

Breadth-First Frog

The *frog problem* can be solved intuitively using a vanilla implementation of *breadth-first search* over indices as nodes.

```
Q = [0], distance = {0: 0}
while (true)
   x = Q.dequeue()
   if (x == n-1) return distance[x]
   for (j from 1 to A[i] if i + j < n)
       if (i+j not in distance)
           distance[i+j] = 1 + distance[i]
           Q.enqueue(i+j)
```

Remark. The queue `Q` can be removed for an *improved solution*, as only increasing integers are enqueued.

Improved Breadth-First Frog

The *frog problem* can be solved more efficiently using an adapted implementation of *breadth-first search*, avoiding a stack.

```
furthest = 0, distance = {0: 0}
for (i = 0; furthest < n - 1; i++)
   if (i + A[i] > furthest)
       for (j from furthest to i + A[i])
           distance[i+j] = distance[i] + 1
       furthest = i + A[i]
return distance[n-1]
```

Remark. The `distance` map can be avoided as well by counting the ”waves” or breadth layers, for a *linear solution*.

Linear Frog Solution

The *frog problem* can be solved linearly by improving the *breadth-first search solution* to simply count the waves or breadth layers.

```
furthest = 0, jumps = 0, cur_wave = 0
for (i = 0; i < n - 1; i++)
   furthest = max(furthest, i + A[i])
   if (i == cur_wave)
       jumps++
       cur_wave = furthest
return jumps
```

Remark. This solution can not produce the jump sequence.

Closest Stars Problem

Given an array `A` of 3-dimensional coordinates (x, y, z) for n stars, determine the k closest stars to the center of the universe $(0, 0, 0)$, where $k \ll n$.

```
input: [(123.8, 86.3, 912.5), ... ×10^12 ], k = 10
solution: [(54.6, 71.5, 9.1), ...]
```

Solution ↓ .

Closest Stars Solution

The *closest stars* problem can be solved in $O(n \log k)$ by using a max heap of maximum size k while iterating over the array `A` and progressively replacing the current max with lower stars.

```
H = max-heap()
for (star in A)
    if (H.size() < k) H.insert(star)
    else if (H.top() > star)  // compare distance
        H.pop(); H.insert(star)
```

Linked List to Binary Search Tree

Given the head to a sorted singly-linked list, return the root of the corresponding balanced binary search tree.

```
input: [-10, -3, 0, 5, 9]              // head (-10)
solution: [0, -3, 9, -10, null, 5]  // array repr.
```

Solution ↓ .

LL to BST Solution

A sorted array can be *converted* to a balanced binary search tree by recursively applying the transformation to two subarrays of equal length. To determine the middle of a linked-list we use the *Tortoise and the Hare* method.

```
toBST(head, tail)
   if (head == tail) return NULL
   slow = fast = head
   while (fast != tail && fast.next != tail)
       slow = slow.next; fast = fast.next.next  // *
   root = Node(slow.value)
   root.left = toBST(head, slow)
   root.right = toBST(slow.next, tail)
   return root
toBST(root, NULL)  // for solution
```

k-Sum Tree Paths

Given the root of a binary tree, determine all paths from the root to a leaf with sum equal to a given k.

```
input: root, k = 22
solution: [[5, 4, 11, 2], [5, 8, 4, 5]]
```

Solution ↓ .

k-Sum Tree Paths Solution

The *k-sum tree paths* problem can be solved recursively using a single auxiliary list `c = []` and the following helper function.

```
helper(node, k, c, solution)
    if (node == NULL) return
    c.push_back(node)
    if (node.is_leaf() and node.val == k)
        result.push_back(c) // copy!
    else
        helper(node.left, k - node.val, c, solution)
        helper(node.left, k - node.val, c, solution)
    c.pop_back()
```

Lonely Number Problem

Given an unordered array `A` of integers, determine the only element which does not appear twice.

```
input: [3, 5, 2, 1, 4, 3, 1, 5, 4]
solution: 2
```

Solution ↓ .

XOR Reduction

The *lonely number problem* can easily be solved in linear time and without additional memory by performing a bitwise XOR reduction over the array, since `a^a = 0` and `a^0 = a`.

```
// python & C++
reduce(lambda x, y: x ^ y, A)
accumulate(A.begin(), A.end(), 0, bit_xor<int>());
```

Trapped Rain Water

Given an array `height` of non-negative integers representing an elevation map, compute how much water it is able to trap.

```
input: [0,1,0,2,1,0,1,3,2,1,2,1]
solution: 6
      #
  #ooo##o#
#o##o######
```

Solutions ↓ ↓ ↓ .

Trapped Rain Water DP

The *trapped rain water* problem can be solved using dynamic programming. The water trapped at every position `i` can simply be computed using the highest bars to its left and right.

```
for (i from 1 to n)
    leftmax[i] = max(leftmax[i-1], h[i])
for (i from n to 1)
    rightmax[i] = max(rightmax[i+1], h[i])
for (i from 1 to n)
    water += min(leftmax[i], rightmax[i]) - h[i]
```

Remark. We here think *vertically* instead of horizontally. For the latter, see this *solution* using stacks.

Trapped Rain Water Stack

The *trapped rain water* problem can be solved using a stack onto which we push every position `i` and then retroactively flood them when encountering higher terrain.

```
for (i from 1 to n)
    while (stack and h[stack.top] < h[i])
        t = stack.pop()
        distance = i - stack.top() - 1
        height_diff = min(h[i], h[stack.top()]) - h[t]
        water += distance * height_diff
    stack.push(i)
```

Remark. The stack always contains decreasing heights. Similar to the *next greater element* solution.

Merge k Sorted Lists

Merge k sorted linked lists and return it as one sorted list.

```
input: [1->4->5, 1->3->4, 2->6]
solution: 1->1->2->3->4->4->5->6
```

Solutions ↓ ↓ .

Merge Sorted Lists with Max Heap

We can *merge k sorted lists* by maintaining k pointers to the beginning of each list and progressively picking the lowest element. The latter operation can be optimized using a *min heap* of size k.

```
h = min-heap
for (head in list-heads) h.add(head, head.val)
while h not empty:
    node = h.pop()
    copy node to new list
    if (node.next) h.add(node.next, node.next.val)
```

Remark. A more space-efficient *solution* also exists.

Divide and Merge Sorted Lists

We can *merge k sorted lists* using divide-and-conquer by successively merging pairs of lists in place.

```
amount = len(lists), interval = 1
while true:
    for (i from 0 to amount - interval by interval * 2)
         merge2lists(lists[i], lists[i+1])
    interval *= 2
```

Remark. linear and in-place `merge2lists` left as an exercise.

Largest Rectangle in Histogram

Given an array `h` of bar heights for a histogram, determine the largest area of a rectangle contained inside `h`.

```
input: [2, 1, 5, 6, 2, 3]
solution: 10
```

Solution ↓ .

Linear Largest Rectangle

To find the *largest rectangle* in a histogram we determine for every bar the first smaller bars on either side. We push every element onto a stack, and pop them when we encounter a smaller bar `i`. Then bar `s.top` is bounded by `s.top.top` and `i`.

```
s = stack, h.push_back(0) // sentinel
for (i from 1 to n)
    while (s and h[s.top] >= h[i])
        height = h[s.pop()]
        left = s ? s.top : -1
        largest = max(largest, height * (i - left - 1))
    s.push(i)
```

Remark. The stack `s` always contains increasing bars. Similar to the *next greater element* or *trapped rain water* solution.

Binary Tree Maximum Path

Given the `root` to a non-empty binary tree, find the path between any two nodes with maximum sum.

```
input: [-10, 9, 20, null, null, 15, 7]
solution: 42  // 15->20->7
```

Solution ↓ .

Binary Tree Maximum Path Recursion

We can determine the *path with maximum sum* in a binary tree recursively, using a function returning the maximum path *ending* in a given node while also maintaining a *global* maximum.

```
maxSumToNode(node):
    if (node == NULL) return 0
    left = max(0, maxSumToNode(node->left))
    right = max(0, maxSumToNode(node->right))

    solution = max(solution, left + right + node->val)
    return max(left, right) + node->val
```

Remark. Use common dynamic programming subtree optimality.

Longest Consecutive Sequence

Given an unsorted array `A` of integers, determine the length of the longest arbitrary sequence of consecutive elements.

```
input: [4, 8, 1, 6, 3, 9, 2]
solution: 4  // [1, 2, 3, 4]
```

Solution ↓ .

Longest Consecutive Sequence Solution

We can determine the length of the *longest consecutive sequence* of an array `A` linearly by creating a set of all elements, before counting all successors for every element which has to be the *beginning* of some sequence.

```
 s = set(A), longest = 0
 for (i in A)
    if (!s.contains(i-1)) // beginning!
        streak = 1
        next = i + 1
        while (s.contains(next++)) streak++
        longest = max(longest, streak)
return longest
```

Bursting Balloons

Given an array `B` of balloons, bursting balloon `i` yields `B[left]` × `B[i]` × `B[right]` coins, where `left` and `right` are its neighbors. Determine the maximum sum of coins achievable.

```
input: [3,1,5,8]
solution: 167  // 3*1*5 + 3*5*8 + 3*8 + 8
```

Solution ↓ .

Bursting Balloons DP

The *bursting balloons* problem can be solved using dynamic programming, using `dp[i][j]` as the maximum coins from bursting balloons in *range* `i..j`. We divide each range using the *last* balloon to burst, which will yield `B[i-1]` × `B[last]` × `B[j+1]` coins.

```
// add sentinel 1 around B
for (k from 1 to n)  // length of range
  for (left from 1 to n-k+1)
    right = left + k
    for (last from left to right)
      coins = B[left-1] * B[last] * B[right+1]
      dp[left][right] = max(dp[left][right],
        coins + dp[left][last-1] + dp[last+1][right])
```

Solution, Dynamic Programming, Divide and Conquer, Bursting Balloons, Range

Median of Sorted Arrays

Determine the median of two sorted arrays `A` and `B` of size n and m respectively in runtime complexity $\log(n+m)$.

```
input: [1, 3, 5, 8, 9], [0, 2, 4, 6, 7]
solution: 4.5  // 0,1,2,3,4 - 5,6,7,8,9
```

Solution ↓ **.**

Median of Sorted Arrays Solution

The *median of two sorted arrays* can be found by searching the element $k = (n + m)/2$. Compare element $k/2$ of each array. If `A[k] > B[k]` the median can't be in `B[..k]`. Discard the correct subarray and repeat with element $k - k/2$.

```
findK(k, A, B)
    ...
```

Sorted Matrix Search

Given a two-dimensional $m \times n$ matrix `M` such that each row and column is sorted, determine whether `M` contains the target `t`.

```
input: [ [1, 4, 7],
         [2, 8, 9],
         [4, 6, 11] ],  target = 8
solution: true
```

Solution ↓ .

Linear Sorted Matrix Search

To search an element in a *sorted matrix* `M` in linear $O(m+n)$ runtime, begin from the bottom left corner and move each axis in only one direction, according to `M[x][y]` and `target`.

```
x = m, y = 0
while (x >= 0 and y <= n)
    if (M[x][y] > target) x--        // down one row
    else if (M[x][y] < target) y++   // right one col
    else return true
```

Remark. Think about why we can't miss `target`.

Uniform Stream Sampling

Given a stream S of unknown length, produce a uniform random sample of size k with limited $O(k)$ storage.

```
input: S = [2, 4, 7, 9, ... ×10^100, 8, 5, 1], k = 5
solution: [5, 7, 3, 1, 2]
```

Solution ↓ .

Reservoir Sampling

To produce a *uniform sample from a large stream* with limited memory, we apply the *reservoir sampling* or `R` algorithm: replace a random element from the reservoir with every new element s_i with probability $p = k/i$.

```
for i from k+1 to n
    j = random(1, i)  // inclusive
    if (j <= k) R[j] = S[i]
```

Before s_i, elements are in the reservoir with probability $k/(i-1)$. Multiplying this by the probability of remaining selected we get:

$$p = \frac{k}{i-1}\left\{\frac{i-k}{i} + \frac{k}{i}\cdot\frac{k-1}{k}\right\} = \frac{k}{n+1}$$

Reverse Interleave Linked List

Given a singly linked list `L`, reorder it by interleaving elements from each end as follows:

```
input: L = [l0, l1, l2, ..., lx, ly, lz]
solution: [l0, lz, l1, ly, l2, lx, ... ]
```

Solution ↓ .

Reverse Interleave Linked List Solution

A linked list `L` can be *reversely interleaved* by simply reversing the second half found using the *Tortoise and the Hare* method, before merging the two halves.

```
slow = root, fast = root
while (fast != NULL && fast.next != NULL)
    slow = slow.next, fast = fast.next.next
end = reverse(slow)  // end points to lz
return merge(root, end)
```

Remark. Linear in-place `reverse` and `merge` left as exercise. A similar approach can determine if `L` is *was* a palindrome.

Solution, Linked List, Merge, Reverse, Interleave, Tortoise and the Hare

Binary Tree Levels

Given the `root` of a binary tree, return a list of its levels or layers.

```
input: [3, 9, 20, ∅, ∅, 15, 7]
solution: [[3], [9, 20], [15, 7]]
```

Solution ↓ .

Binary Tree Level Traversal

The *levels of a binary tree* can be produced by using *breadth-first search* with an additional internal loop over every level.

```
results = [] // if root != null
q = queue(root)
while queue:
   layer = []
   size = q.size()
   for i from 1 to size: // iterate over layer
       x = q.pop()
       layer.push(x)
       if (x.left) q.push(left)
       if (x.right) q.push(right)
   results.push(layer)
```

Remark. Alternatively remember the last node of each layer.

Binary Tree Right View

Given the `root` of a binary tree, return its right view, i.e. the right-most node of every level

```
input: [3, 9, 20, 5, ∅, 15, 7, 1]
solution: [3, 20, 7, 1]
```

Solution ↓ .

Binary Tree Right View Traversal

To determine the *right view of a binary tree* we use a a *level traversal* and output the last element of each layer.

```
q = queue(root)
while queue:
   s = q.size()
   for i from 1 to s:     // layer loop
       x = q.pop()
       if i == s: results.push_back(x) // last element
       if (x.left) q.push(x.left)
       if (x.right) q.push(x.right)
```

Symmetric Binary Tree

Given the `root` to a binary tree, determine whether the tree is symmetric along its vertical center axis, i.e. a mirror of itself.

```
input: [1, 2, 2, 3, 4, 4, 3], [1, 2, 2, 0, 3, 0, 3]
solution: true, false
```

Solution ↓ .

Recursive Binary Tree Symmetry

To determine whether a *binary tree is symmetric*, we use recursion over symmetric paths to the leaves.

```
bool symmetric(left, right)
    if (left == null and right == null) return true
    if (left == null or right == null) return false
    if (left.value != right.value) return false

    return (symmetric(left.left, right.right) &&
            symmetric(left.right, right.left))
```

Lowest Common Ancestor

Given the `root` to a *binary tree* and two nodes `p` and `q`, determine their lowest common ancestor.

```
input:  [3, 5, 1, 6, 2, 0, 8, ∅, ∅, 7, 4], p = 7, q = 6
solution: 5
```

Solution ↓ .

Recursive Lowest Common Ancestor

To determine *lowest common ancestor* in a binary tree, we use recursive calls to each nodes children.

```
lca(n, p, q):
    if (n == null || n = q || n = q) return n
    left  = lca(n->left, p, q)
    right = lca(n->right, p, q)

    if (left and right) return n
    return (left) ? left : right
```

Remark. Determining the LCA in a *search* tree is trivial.

Validate Postorder

Given an array of nodes `A`, determine whether it corresponds to a valid postorder traversal of some binary tree.

```
input: [9,3,4,#,#,1,#,#,2,#,6,#,#]
solution: true
```

Solution ↓ .

Validate Postorder Algorithm

To determine whether `A` is a *validate postorder* traversal, we use a stack `s` which progressively replaces leaves by `#`.

```
for n in nodes:
    if (n == "#")
        while (s and s[-1] == "#" and s[-2] != "#")
            s.pop(); s.pop()
    s.push(n)
 return s == ["#"]
```

Remark. Alternatively count the in- and out-degree difference.

Sum Root to Leaf Numbers

Given the `root` to a binary tree of nodes `0-9`, every path p from the root to a leaf defines a number. Return their sum.

```
input: [1, 2, 3, 1, ∅, 5, 9]
solution: 395 // 121 + 135 + 139
```

Solution ↓ .

Sum Roof to Leaf Numbers Solution

To determine the *sum of root-to-leaf numbers* we use simple *depth-first* recursion with an accumulator.

```
sum(node, acc):
    if (node == NULL) return 0
    if (node is leaf) return acc + node.val

    acc = (acc + node.val ) * 10
    return sum(node.left, acc) + sum(node.right, acc)
```

Remark. `acc` could also just be a reference, which we would need to restore it to its original value at the end of each call.

Minimum Height Tree

Given an undirected acyclic graph with `n` nodes and a list of edges `(i, j)` determine all nodes which, if taken to be the root, result in a tree with minimum height.

```
input: n = 4, e = [[1, 0], [1, 2], [1, 3]]
solution: [1]  // height of 1
```

Solution ↓ .

Minimum Height Tree Algorithm

To determine the *minimum height trees* we iteratively identify all leaves and delete them. The last one or two nodes that remain must be the optimal roots.

```
while (graph.size > 2)
   remove all leaves
return graph.nodes
```

Two Stock Transactions

Given an array where `p[i]` is the price of some stock on day `i`, maximize the profit with *two non-overlapping* transactions.

```
input: [3,3,5,0,0,3,1,4]
solution: 6  // 0->3 + 1->4
```

Solution ↓ .

Linear Two Stock Optimization

Determine the maximum profit of *two stock transactions* by building `left[i]` and `right[i]` representing the maximum profit with a single transaction respectively before and after day `i`.

```
highest_price = p[n-1]
for (i from n-2 to 0)
   highest_price = max(p[i], highest_price)
   right[i] = max(right[i+1], highest_price - p[i])

lowest_price = p[0]
for (i from 1 to n-1)
   lowest_price = min(p[i], lowest_price)
   left[i] = max(left[i-1], p[i] - lowest_price)

return max(left[i] + right[i] for i from 0 to n-1)
```

Remark. This also solves the *single stock transaction* problem. With *unlimited* stock transactions buy at lows, and sell at highs.