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
}
```

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.

Mergesort

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

Mergesort is a stable sorting algorithm recursively sorting two subarrays 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* and finding strongly connected *components*.

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^{T}.

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

Remark. G^{T} 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.

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 = ∞ for (u in V) but s.d = 0
S = ∅ // 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 ∪ { 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.

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 $\{1, ..., k\}$.

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

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] ≥ 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 = -∞
    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 $Y_{1..j}$.

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

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}.

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++
```

Greedy, Knapsack, Solution

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 fᵢ
S = {A[0]}
f = f[0]
for (i from 1 to n)
    if (s[i] ≥ f)
        S = S ∪ {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] ≤ s[i]) break
    assign i to j
    r[j] = f[i]
```

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 $\boxed{\downarrow}$.

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.

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.