Summary
CS-250 Algorithms

Clément Charollais

Dernière mise à jour : January 28, 2019

Contents

1 Loop Invariant

Sort of induction proof ù

- Initialisation : Property true prior entry of a loop ($P(a)$)
- Maintenance : Proving that property still holds for one more element after the iteration ($P(n) \Rightarrow P(n+1)$)
- Termination : Proving that after termination the property still holds ($P(N)$)

2 Sorting

Any comparison based sorting algorithm has a time complexity of $\Omega(n \log n)$

2.1 Insertion Sort

Iterates through the list and places the $j-th$ element at the correct place in the already sorted subarray $A[0 \dots j-1]$.

Time complexity	:	$O(n^2)$ (reverse sorted n comparisons and n swaps), $\Omega(n)$ (already sorted: only n comparisons)
Space complexity	:	$\Theta(n)$ total $\Theta(1)$ auxiliary
In-place	:	Yes

Algorithm 1 Insertion Sort

```
function INSERTION-SORT($A, n$)                ▷ Sorts the array $A$ of size $n$
    for $j = 2$ to $n$
        $key = A[j]$
        $i = j - 1$
        while $i > 0$ and $A[i] > key$ do
            $A[i+1] = A[i]$
            $i = i - 1$
        $A[i+1] = key$
```

2.2 Merge Sort

Splits the array into two half size arrays recursively and only sorts when the problem becomes trivial or small enough to be bruteforced.

Time complexity	:	$T(n) = 2T(\frac{n}{2}) + \Theta(n) = \Theta(n \log n)$
Space complexity	:	$\Theta(n)$total, $\Theta(n)$ auxiliary
In-place	:	No

Algorithm 2 Merge Sort

```
function MERGE-SORT($A, p, r$)        ▷ Sorts the sub-array starting at $p$ and ending at $r$
    if $p < q$ then                                  ▷ Checks for base case
        $q = \lfloor \frac{r+p}{2} \rfloor$                              ▷ divide
        MERGE-SORT($A, p, q$)                              ▷ conquer
        MERGE-SORT($A, q, r$)                              ▷ conquer
        MERGE($A, p, q, r$)                                ▷ combine
```

Algorithm 3 Merge

```
function MERGE($A, p, q, r$)                ▷ Combines arrays $A[p \dots q]$ and $A[q \dots r]$
    $n_1 = q - p + 1$
    $n_2 = r - p$
    Let $L[1 \dots n_1 + 1], L[1 \dots n_1 + 1]$ be new arrays
    for $i = 1$ to $n_1$
        $L[i] = A[p + i - 1]$
    for $j = 1$ to $n_2$
        $L[j] = A[q + j]$
    $L[n_1 + 1] = \infty$
    $R[n_2 + 1] = \infty$
    $i = 1$
    $j = 1$
    for $k = p$ to r
        if $L[i] \leq R[j]$ then
            $A[k] = L[i]$
            $i = i + 1$
        else
            $A[k] = R[j]$
            $j = j + 1$
```

2.3 HeapSort

Heapsort uses Max heaps to sort an array, To do so, it builds a max heap containing the array's keys. It then extracts the maximum of the heap and swaps it with the last element of the heap, and runs MAX-HEAPIFY on the heap, this time on a heap of one element less (in order to leave the maximum at the end of the array)

Time complexity	:	$O(n \log n)$
Space complexity	:	$\Theta(n)$ total $\Theta(1)$ auxiliary
In-place	:	Yes

Algorithm 4 HeapSort

```
function HEAPSORT(n)
    BUILD-MAX-HEAP($A, n$)
    for i = n downto 2
        SWAP($A, i, 1$)
        MAX-HEAPIFY($A, 1, i - 1$)
```

2.4 QuickSort

Randomized version of a divide and conquer algorithm. Here the complex part is the splitting. The divide step for a set S is to split it into T and U by picking a pivot p at random in S and to construct T and U such that $e \in T \Leftrightarrow e \leq p$ and $e \in U \Leftrightarrow a > p$.

Time complexity	:	Expected time $O(n \log n)$
Space complexity	:	$\Theta(n)$
In-place	:	Yes

Algorithm 5 QuickSort
```
function QuickSort($A, p, r$)
    if $p < r$ then
        $q$ = Partition($A, p, r$)
        QuickSort($A, p, q - 1$)
        QuickSort($A, q + 1, r$)
function Partition($A, p, r$)        ▷ Chooses arbitrary pivot,reorganizes $A$ and returns position of the pivot
    $i$ = Random($p, r$)
    Swap($A, r, i$)
    $x = A[r]$
    $i = p - 1$
    for $j = p$ to $r - 1$
        if $A[j] \leq x$ then
            $i = i + 1$
            Swap($A, i, j$)
    Swap($A, i + 1, r$)
    return $i + 1$
```

The expected time complexity is low because the expected total number of comparisons between two elements is described as follows

$$\begin{aligned}\mathbb{E}[X] &= \mathbb{E}\left[\sum_{i=1}^{n}\sum_{j=i+1}^{n} P[z_i \text{ is compared to } z_j]\right] \\ &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} P[z_i \text{ is compared to } z_j] \\ &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \frac{2}{k+1} \\ &< \sum_{i=1}^{n-1}\sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log n) = O(n \log n)\end{aligned}$$

2.5 Linear time sorting

It is however possible to beat the $\Omega(n \log n)$ lower bound if the elements have a certain structure. Comparaison sort can sort n elements $e \in \{0, 1, 2, \dots, k\}$ in $\Theta(n + k)$, but this algorithm gets impractical for large values of k.

3 Data Structures

3.1 Doubly Linked List

List-Insert(L, x)	:	$\Theta(1)$
List-Delete(L, x)	:	$\Theta(1)$
List-Search(L, k)	:	$\Theta(n)$

Implementation using a sentinel cell (*nil*) with no value pointing to the first element and that is pointed by the last element of the list as its next cell. Each other cell contains a pointer to the next cell(*next*), to the previous cell (*prev*) and its value.(*key*)

Algorithm 6 List-Insert
```
function List-Insert($L, x$)
    $x.next = L.nil.next$
    $L.nil.next.prev = x$
    $L.nil.next = x$
    $x.prev = L.nil$
```

Algorithm 7 List-Delete

```
function LIST-DELETE($L, x$)
    $x.prev.next = x.next$
    $x.next.prev = x.prev$
```

Algorithm 8 List-Search

```
function LIST-SEARCH($L, k$)                    ▷ Returns $L.nil$ if the key is not in the list
    $x = L.nil.next$
    while $x \neq L.nil$ and $x.key \neq k$ do
        $x = x.next$
    return $x$
```

3.2 Stacks (LIFO)

STACK-EMPTY(S)	:	$\Theta(1)$
PUSH(S, x)	:	$\Theta(1)$
PEEK(S)	:	$\Theta(1)$
POP(S)	:	$\Theta(1)$

Implementation using Arrays, with a pointer to the top of the stack (*top*).

Algorithm 9 Stack-Empty

```
function STACK-EMPTY($S$)
    if $S.top = 0$ then
        return true
    else
        return false
```

Algorithm 10 Push

```
function PUSH($S, x$)
    $S.top = S.top + 1$
    $S[S.top] = x$
```

Algorithm 11 Peek

```
function PEEK($S$)
    if STACK-EMPTY($S$) then
        error: Stack empty
    else
        return $S[S.top]$
```

Algorithm 12 Pop

```
function POP($S$)
    if STACK-EMPTY($S$) then
        error: Stack underflow
    else
        $S.top = S.top - 1$
        return $S[S.top + 1]$
```

3.3 Queues (FIFO)

ENQUEUE(Q, x)	:	$\Theta(1)$
DEQUEUE(Q)	:	$\Theta(1)$

Implementation using Arrays, with a pointer to the end of the queue (*tail*)and a pointer to the first element of the queue (*head*).

Algorithm 13 Enqueue

```
function ENQUEUE($Q, x$)
    $Q[Q.tail] = x$
    if $Q.tail = Q.length$ then
        $Q.tail = 1$
    else
        $Q.tail = Q.tail + 1$
```

Algorithm 14 Dequeue

```
function DEQUEUE($Q, x$)
    $x = Q[Q.head]$
    if $Q.head = Q.length$ then
        $Q.head = 1$
    else
        $Q.head = Q.head + 1$
    return $x$
```

3.4 Heaps - Priority Queues

MAX-HEAP-ROOT(A)	:	$\Theta(1)$
MAX-HEAP-LEFT-CHILD(i)	:	$\Theta(1)$
MAX-HEAP-RIGHT-CHILD(i)	:	$\Theta(1)$
MAX-HEAP-PARENT(i)	:	$\Theta(1)$
MAX-HEAPIFY(A, i, n)	:	$\Theta(\log h) = O(\log n)$
BUILD-MAX-HEAP(A, n)	:	$O(n)$
INSERT(A, k, n)	:	$O(\log n)$
MAXIMUM(A)	:	$O(1)$
EXTRACT-MAXIMUM(A, n)	:	$O(\log n)$
INCREASE-KEY(A, x, k)	:	$O(\log n)$

A heap is a nearly complete binary tree. (Max) Heaps maintain the (Max) Heap property. The key of i's children are smaller or equal than i's key. In memory, they are stored by layer, with this storing method we get that.

Algorithm 15 Max-Heap-Root

```
function MAX-HEAP-ROOT
    return $A[1]$
```

Algorithm 16 Max-Heap-Left-Child

```
function MAX-HEAP-LEFT-CHILD($i$)
    return $2i$
```

Algorithm 17 Max-Heap-Right-Child

```
function MAX-HEAP-RIGHT-CHILD($i$)
    return $2i + 1$
```

Algorithm 18 Max-Heap-Parent

```
function MAX-HEAP-PARENT($i$)
    return $\lfloor \frac{i}{2} \rfloor$
```

In in order to maintain the heap property we need a procedure to do so. This procedure starts at node i and if necessary, swaps $A[i]$ with the largest of its two children and continues swapping down until the tree rooted at i is a max-heap.

NB: this procedure only guarantees the max-heap property if both its children are max heaps before starting the procedure.

The worst case runtime of this procedure is the height of the node at which it is called i.e. $O(\log n)$

Algorithm 19 Max-Heapify

```
1: function MAX-HEAPIFY($A, i, n$)
2:     $l = \text{LEFT}(i)$
3:     $r = \text{RIGHT}(i)$
4:     if $l \leq n$ and $A[l] > A[i]$ then
5:         $largest = l$
6:     else
7:         $largest = i$
8:     if $r \leq n$ and $A[r] > A[Largest]$ then
9:         $largest = r$
10:    if $largest \neq i$ then
11:        SWAP($A, i, largest$)
12:        MAX-HEAPIFY($A, largest, n$)
```

To build a max heap, it is necessary to call MAX-HEAPIFY from all parents from the lowest parent to the root of the tree. This yields a time of $\sum_{h=0}^{\lfloor \log n \rfloor} \frac{n}{2^h} O(h) = O\left(n \sum_{h=1}^{\infty} \frac{h}{2^h}\right) = O(n)$

Algorithm 20 Build-Max-Heap

```
1: function BUILD-MAX-HEAP($A, n$)
2:     for $i = \left\lfloor \frac{n}{2} \right\rfloor$ downto 1
3:         MAX-HEAPIFY($A, i, n$)
```

Algorithm 21 Insert

```
1: function INSERT($A, k, n$)
2:     $n = n + 1$
3:     $A[n] = -\infty$
4:     INCREASE-KEY($A, n, k$)
```

Algorithm 22 Maximum

```
1: function MAXIMUM($A$)
2:     return $A[1]$
```

Algorithm 23 Extract-Maximum

```
1: function EXTRACT-MAXIMUM($A, n$)
2:     $max = A[1]$
3:     $A[1] = A[n]$
4:     $n = n - 1$
5:     MAX-HEAPIFY($A, 1, n$)
6:     return $max$
```

Algorithm 24 Increase-Key

```
function INCREASE-KEY($A, i, k$)
    if $k < A[i]$ then
        error: The new key is smaller than the current key
    $A[i] = k$
    while $i > 1$ and $A$[PARENT($i$)], $A[i]$ do
        SWAP($A, i$,PARENT($i$))
        $i$ =PARENT($i$)
```

3.5 Binary Search Trees

The useful property of a Binary Search Tree is that for any node, all its descendants in its left subtrees are strictly smaller than itself and all the descendants in the right subtree are greater or equal than itself.

TREE-SEARCH(x, k)	:	$O(h)$
TREE-MINIMUM(x)	:	$O(h)$
TREE-MAXIMUM(x)	:	$O(h)$
TREE-SUCCESSOR(x)	:	$O(h)$
TREE-PREDECESSOR(x)	:	$O(h)$
INORDER-TREE-WALK(x)	:	$\Theta(n)$
PREORDER-TREE-WALK(x)	:	$\Theta(n)$
POSTORDER-TREE-WALK(x)	:	$\Theta(n)$
TREE-INSERT(T, x)	:	$O(h)$
TREE-DELETE(T, x)	:	$O(h)$

Each node stores a pointer to its left child ($left$), right child($right$), parent ($parent$) and value (key). If a child does not exist the sentinel value NIL is placed in its place

Algorithm 25 Tree-Search

```
function TREE-SEARCH($x, k$)                    ▷ Returns NIL if the key is not found
    if $x$ == NIL or $k == x.key$ then
        return $x$
    if $k < x.key$ then
        return TREE-SEARCH($x.left, k$)
    else
        return TREE-SEARCH($x.right, k$)
```

Algorithm 26 Tree-Minimum

```
function TREE-MINIMUM($x$)
    while $x.left \neq$ NIL do
        $x = x.left$
    return $x$
```

Algorithm 27 Tree-Maximum

```
function TREE-MAXIMUM($x$)
    while $x.right \neq$ NIL do
        $x = x.right$
    return $x$
```

The successor of a node x is either the smallest of the numbers in its right subtree of, if it is empty, the node in whose left subtree has x as its maximum. In the second case, it is the first parent that has x in its left subtree. For the predecessor, the argument is the same, but symmetric.

Algorithm 28 Tree-Successor

```
function TREE-SUCCESSOR($x$)
    if $x.right \neq$ NIL then
        return TREE-MINIMUM($x.right$)
    else
        $y = x.parent$
        while $y \neq$ Nil and $x == y.right$ do
            $x = y$
            $y = x.parent$
        return $y$
```

Algorithm 29 Tree-Predecessor

```
function TREE-PREDECESSOR($x$)
    if $x.left \neq$ NIL then
        return TREE-MAXIMUM($x.left$)
    else
        $y = x.parent$
        while $y \neq$ Nil and $x == y.left$ do
            $x = y$
            $y = x.parent$
        return $y$
```

The inorder tree walk yields the sorted sequence of the keys

Algorithm 30 Inorder-Tree-Walk

```
function INORDER-TREE-WALK($x$)
    if $x \neq$ NIL then
        INORDER-TREE-WALK($x.left$)
        PRINT($x.key$)
        INORDER-TREE-WALK($x.right$)
```

Algorithm 31 Preorder-Tree-Walk

```
function PREORDER-TREE-WALK($x$)
    if $x \neq$ NIL then
        PRINT($x.key$)
        PREORDER-TREE-WALK($x.left$)
        PREORDER-TREE-WALK($x.right$)
```

Algorithm 32 Postorder-Tree-Walk

```
function POSTORDER-TREE-WALK($x$)
    if $x \neq$ NIL then
        POSTORDER-TREE-WALK($x.left$)
        POSTORDER-TREE-WALK($x.right$)
        PRINT($x.key$)
```

The idea of insertion is to search T for the value we want to insert and once we reach NIL, we replace NIL with x

Algorithm 33 Tree-Insert

```
function TREE-INSERT($T, z$)
    $y = Nil$
    $x = T.root$
    while $x \neq$ NIL do
        $y = x$
        if $z.key < x.key$ then
            $x = x.left$
        else
            $x = x.right$
    $z.parent = y$
    if $y ==$ Nil then                                  ▷ The tree $T$ was empty
        $T.root = z$
    else if $z.key < y.key$ then
        $y.left = z$
    else
        $y.right = z$
```

For deletion, three cases arise

- The element to be deleted is childless, hence we can just remove it
- The element to be removed has only one child, hence we can just replace it with its child.
- The element to be deleted has two children, hence we need to find its successor and replace it with its successor.

In order to do so we use an auxiliary method transplant which replaces a subtree u with another v.

Algorithm 34 Transplant

```
function TRANSPLANT($T, u, v$)
    if $u.p ==$ NIL then
        $T.root = v$
    else if $u == u.parent.left$ then
        $u.parent.left = v$
    else
        $u.parent.right = v$
    if $v \neq$ NIL then
        $v.p = u.p$
```

Algorithm 35 Tree-Delete

```
function TREE-DELETE($T, x$)
    if $x.left ==$ NIL then                             ▷ $x$ has no left child
        TRANSPLANT($T, x, x.right$)
    else if $x.right ==$NIL then                        ▷ $x$ has only a left child
        TRANSPLANT($T, x, x.left$)
    else                                                ▷ $x$ has two children
        $y =$TREE-MINIMUM($x.right$)                    ▷ $y$ is $x$'s successor
        if $y.parent \neq x$ then
            TRANSPLANT($T, y, y.right$)
            $y.right = x.right$
            $y.right.parent = y$
        TRANSPLANT($T, x, y$)
        $y.left = x.left$
        $y.left.parent = y$
```

3.6 Disjoint Sets

The Disjoint Sets data structure consists in a collection of disjoint dynamic sets. Each is identified by a representative which is some member of the set.

Make-Set(x)	:	$O(1)$
Find(x)	:	$O(\alpha(n))$
Union(x, y)	:	$O(\alpha(n))$, with α being the inverse Ackermann function whose value is smaller than 5 for all practical input size

NB: These runtimes are achieved if both Path compression and Union by rank are used.

The disjoint-sent data structure is represented using a forest, whose tree's root are the representative of the sets and where each node points to its parent.

Algorithm 36 Make-Set

```
function Make-Set($x$)
    $x.p = x$
    $x.rank = 0$
```

Algorithm 37 Find-Set

```
function Find-Set($x$)
    if $x.p\ != x$ then
        $x.p =$ Find-Set($x.p$)
    return $x.p$
```

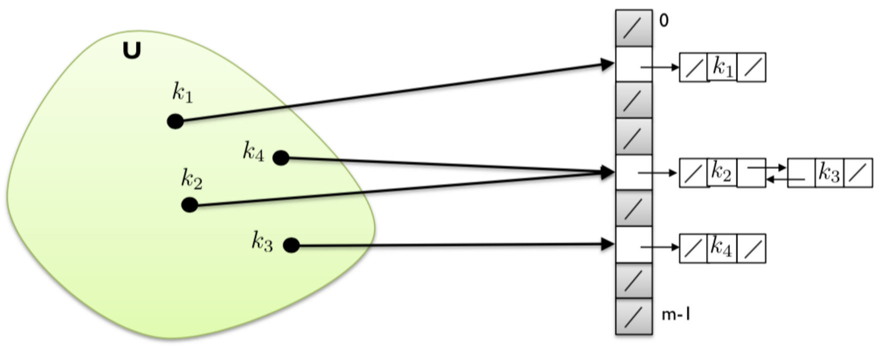
Algorithm 38 Union

```
function Union($x, y$)
    Link(Find-Set($x$),Find-Set($y$))
function Link($x, y$)
    if $x.rank > y.rank$ then
        $y.p = x$
    else
        $x.p = y$
        if $x.rank == y.rank$ then
            $y.rank = y.rank + 1$
```

3.7 Hash Tables

In contrast to Direct access table, where a table is of the size of the universe of the keys U, the table is only of the size of the image space of a Hash function $h : U \mapsto \{0, 1, \dots, m-1\}$ and n items stored are indexed with the hash of the key.

	Direct Access table	Hash table
Space	$O(\lvert U \rvert)$	$m + n$
Search	$O(1)$	Expected $O(1 + \frac{n}{m})$, $O(1)$ if $m = \Theta(n)$
Deletion	$O(1$	$O(1)$
Insertion	$O(1)$	$O(1)$



Hash function Here, simple uniform hashing is assumed, i.e. the hashes are uniformly and independently distributed on the image space of the hash function.

4 Divide and Conquer

The paradigm of Divide-And-Conquer algorithms is the following approach towards the resolution of a problem.

Divide the problem into smaller instances of the same problem

Conquer the subproblems recursively until the subproblem is trivial to solve

Combine the solutions to the subproblems in order to get the solution of the original problem.

The general form of the time complexity of a Divide-and-Conquer algorithm is always

$$T(n) = \begin{cases} \Theta(1) & n \leq c \\ aT(\frac{n}{b}) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Where the problem is split into a subproblems of size $\frac{n}{b}$, the cost of the divide step is $D(n)$ and the cost of the combination step is $C(n)$ and the problem is trivial to solve when its size reaches c

4.1 Master theorem

Let $a \geq 1, b \geq 1$ be constants, $f(n)$ a function, and $T(n)$ be defined on the nonnegative integers by the recurrence.

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ aT(\frac{n}{b}) + f(n) & \text{otw.} \end{cases}$$

Then, $T(n)$ has the following asymptotic bounds:

- If $f(n) = O(n^{\log_b (a-\epsilon)})$ for some $\epsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$
- If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$
- If $f(n) = \Omega(n^{\log_b (a+\epsilon)})$ for some $\epsilon > 0$ and if $af(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then$T(n) = \Theta(f(n))$

4.2 Akra-Bazzi

For recurrence relation $T(n)$ that arise from uneven splits, the master theorem is not applicable anymore. In this case we can use the Akra-Bazzi Method.

$$T(n) = \begin{cases} \Theta(1) & n = \Theta(1) \\ f(n) + \sum_{i=1}^{k} a_i T(\frac{n}{b_i}) & \text{otw.} \end{cases}$$

Where k, $a_i > 0$, and $b_i > 1$ are constants for all i, and $f(n) = \Omega(n^c)$ and $f(n) = O(n^d)$ for some constants $0 < c \leq d$.

$$T(n) = \Theta\left(n^\rho \left(1 + \int_1^n \frac{f(u)}{u^{\rho+1}}\,\mathrm{d}u\right)\right) \qquad \sum_{i=1}^{k} \frac{a_i}{b_i^\rho} = 1$$

4.3 Karatsuba

4.4 Strassen

To multiply two $n \times n$ matrices we can split them each into four submatrices and performing only seven $\frac{n}{2} \times \frac{n}{2}$ matrix multiplications get the result of the full multiplication.

Time complexity : $T(n) = 7T(\frac{n}{2}) + \Theta(n^2) = \Theta(n^{\log_2 7}) \cong \Theta(n^{2.8})$

$$AB = C$$
$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

We can get all of C's components with these formulas

$$\begin{aligned} M_1 &:= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) & M_2 &:= (A_{2,1} + A_{2,2})B_{1,1} \\ M_3 &:= A_{1,1}(B_{1,2} - B_{2,2}) & M_4 &:= A_{2,2}(B_{2,1} - B_{1,1}) \\ M_5 &:= (A_{1,1} + A_{1,2})B_{2,2} & M_6 &:= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\ M_7 &:= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \end{aligned}$$

$$\begin{aligned} C_{1,1} &= M_1 + M_4 - M_5 + M_7 & C_{1,2} &= M_3 + M_5 \\ C_{2,1} &= M_2 + M_4 & C_{2,2} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

5 Graphs

A graph $G = (V, E)$ consists in a set of vertices V and a set of edges E. A graph is undirected if, $(u, v) \in E \Leftrightarrow (v, u) \in E \quad \forall u, v \in V$. A graph is weighted if a function $w : E \mapsto \mathbb{R}$ is assigned to it which maps weights to each of the edges in the graph.

5.1 Representation

5.1.1 Adjacency List

Time to find if $(u, v) \in E$	:	$O(\text{degree}(u))$
Time to find all adjacent vertices to u	:	$\Theta(\text{degree}(u))$
Space	:	$\Theta(V + E)$

The Adjacency List of a graph is an array of lists of size V, one for each element of V. Each sublist is a list of all the other vertices connected to it. It can also contain the weight of the corresponding edges if G is weighted.

5.1.2 Adjacency Matrix

Time to find if $(u, v) \in E$	:	$\Theta(1)$
Time to find all adjacent vertices to u	:	$\Theta(V)$
Space	:	$\Theta(V^2)$

The adjacency matrix A is a $V \times V$ matrix where $a_{i,j} \neq 0 \Leftrightarrow (i, j) \in E$. For a weighted graph, $a_{i,j}$ is the weight of (i, j). For an undirected graph, A is symmetric

5.2 BFS

BFS(E, V)	:	$O(E + V)$

Input: $G = (V, E)$, an unweighed either directed or undirected graph and a source vertex $s \in V$

Output: A function $d : V \mapsto \mathbb{R}$ representing the distance from s to all vertices in V

Algorithm 39 BFS

```
function BFS($E, V, s,$)
    Let $Q$ be a new empty queue
    Let $d[0 \dots V]$ be a new array                ▷ $d[i]$ is the distance from $s$ to $i$-th vertex
    ENQUEUE($s$)
    for $v$ in $V$
        $d[v] = \infty$
    $d[s] = 0$
    while !ISEMPTY($Q$) do
        $c =$ DEQUEUE($Q$)
        for $e$ in $E$
            if $(c, e) \in$Adjacency-list$(u)$ and $d[c] == \infty$ then
                ENQUEUE($e$)
                $d[e] = d[c] + 1$
```

5.3 DFS

DFS(E, V) : $O(E + V)$

Input: $G = (V, E)$, an unweighed either directed or undirected graph and a source vertex $s \in V$

Output: Two timestamps v_d and v_f for the discovery time and the finishing time.

Algorithm 40 DFS

```
function DFS($E, V, s,$)
    Let $c[V]$ be a new array      ▷ $c[i]$ is 0 if $i$ has not been discovered, 1 if not finished, 2 otherwise
    Let $d[V]$ be a new array                                  ▷ $d[i]$ is the discovery time of $i$
    Let $f[V]$ be a new array                                  ▷ $f[i]$ is the finishing time of $i$
    for $v \in V$
        $c[v] = 0$
    $time = 0$
    for $v \in V$
        if $c[v] == 0$ then
            DFS-VISIT($E, V, v$)
function DFS-VISIT($E, V, v$)
    $time = timee + 1$
    $d[v] = time$
    $c[v] = 1$
    for $u \in Adj(u)$
        if $c[u] == 0$ then
            DFS-VISIT($E, V, u$)
    $c[v] = 2$
    $time = time + 1$
    $f[v] = time$
```

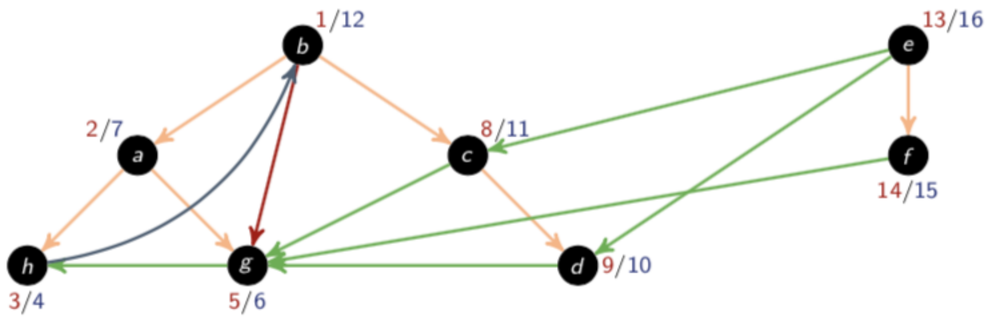
5.3.1 Edges classification

Tree edge In the depth-first forest found by exploring (u, v)

Back edge (u, v) where u is a descendent of v

Forward edge (u, v) where v is a descendent of u, but is not in the tree edge

Cross Edge Any other edge



Parenthesis Theorem $\forall u, v \in V$, exactly one of the following holds

1. $u.d < u.f < v.d < v.f \Leftrightarrow$ neither are descendent of each other
2. $u.d < v.d < v.f < u.f \Leftrightarrow v$ is a descendent of u

5.4 Topological Sort

TOPOLOGICAL-SORT(E, V) $O(E+V)$

Input : Acyclic directed graph

Output : An ordering of the graph with no back edges.

Algorithm 41 Topological-Sort

```
function TOPOLOGICAL-SORT$(E, V)$
    Compute DFS$(E, V, s)$                    ▷ $s$ is an arbitrary vertex
    Output vertices in decreasing finishing time.
```

5.5 Connected Components

CONNECTED-COMPONENTS(E, V) : $O((E+V)\alpha(V)) \cong O(E+V)$

Algorithm 42 Connected-Components

```
function CONNECTED-COMPONENTS$(E, V)$
    for $v \in V$
        MAKE-SET$(v)$
    for $(u, v) \in E$
        if FIND-SET$(u)$ != FIND-SET$(v)$ then
            UNION$(u, v)$
```

For an undirected graph $G = (E, V)$, the vertices u, v are in the same component if there exists a path between them.

5.6 Strongly connected component

STRONGLY-CONNECTED-COMPONENTS(E, V) : $O(E+V)$

A strongly connected component is the **maximal** set $F \subseteq E$ such that all components in F are reachable from each other.

Algorithm 43 Strongly-Connected-Components

```
1: function STRONGLY-CONNECTED-COMPONENTS($E, V$)
2:     DFS($E, V, s$)                                        ▷ $s$ is an arbitrary vertex
3:     Compute $E^T = \{(u, v) : (v, u) \in E\}$              ▷ $E^T$ is the graph with the inverse edges
4:     DFS($E^T, V, t$) by decreasing finishing time.         ▷ $t$ is the last vertex to finish in previous DFS
5:     Output vertices in three of the depth first forest formed in the second DFS as separates SCCs
```

5.7 Flow networks

A flow network is a positively weighted graph with two special vertices, the source s an the sind t. No anti-parallel edges can be assumed WLOG. The weights of the edges are call the capacity of an edge.

5.7.1 Flow

A flow is a function $f : V \times V \mapsto \mathbb{R}$ satisfying the capacity constraint ($\forall u, v \in V$, $0 \le f(u, v) \le c(u, v)$) and flow conservation ($\forall u \in V \setminus \{s, t\}$, $\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$)

The value of a flow $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$

5.8 Ford-Fulkerson - Max-Flow/Min-Cut

FORD-FULKERSON-BFS(E, V)	:	$O(E \cdot V)$
FORD-FULKERSON-FATTEST-PATH(E, V)	:	$O(E + \log(E \cdot U))$, where U is the maximum flow value

The Fold-Fulkerson method can be used to derive the maximum flow through a Flow network and its minimum cut. The maximum flow is simply f after running the algorithm and the minimum cut T, S s.t. $T = V \setminus S$ can be found by defining S to be all vertices reachable from the source in the residual network after the algorithm termination. Th value of the max flow is equal to the value of the flow through the min-cut.

Residual network Given a flow f on a graph $G = (E, V)$, the residual consists of edged representing how the flow on t can change

$$\text{Residual capacity } = c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & (u, v) \in E \\ f(v, u) & (v, u) \in E \\ 0 & \text{otw.} \end{cases}$$

$$\text{Residual Network } = G_f(V, E_f) \text{ where } E_f = \{(u, v) \in V \times V \; c_f(u, v) > 0\}$$

Algorithm 44 Ford-Fulkerson

```
1: function FORD-FULKERSON($E, V$)
2:     Initialize flow $f$ to 0
3:     while $\exists$ an augmenting path $p$ from $s$ to $t$ in Residual Network $G_f$ do
4:         augment $f$ along $p$
5:     return $f$
```

5.9 Min Spanning Trees

PRIM(E, V)	:	$O(E \log(V))$, can be made $O(V \log V)$ with careful queue implementation
KRUSKAL(E, V)	:	$O(E \log(V))$, $O(E\alpha(V))$ if the edges are already sorted

Input An undirected graph $G = (E, V)$ with weight $w : E \mapsto \mathbb{R}$

Output A minimum spanning tree of the graph. A minimum spanning three is a subset T of E that connects all vertices such that $\sum_{e \in T} w(e)$ is minimal. A minimum spanning three is hence an acyclic graph.

5.9.1 Prim

The idea behind Prim algorithm is to greedily grow a tree from an arbitrary vertex adding only the smallest weighted edge that cross the cut defined by partition of the vertices already in the tree and those still not added. It stops when all vertices have been added.

Algorithm 45 Prim

```
function PRIM($E, V, w$)
    Let $H$ be a new Min-Heap
    for $u \in V$
        $u.key = \infty$
        $u.pred =$ NIL
    DECREASE-KEY($H, r, 0$)                                   ▷ Arbitrary $r \in E$
    while ! ISEMPTY($H$) do
        $u =$ EXTRACT-MIN($H$)
        for $v \in Adj[v]$
            if $v \in H$ and $w(u, v), v.key$ then
                $v.pred = u$
                DECREASE-KEY($H, v, w(u, v)$)
```

5.9.2 Kruskal

The idea of this methd is to maintain a forest T that will eventually become thge MST at the end. We gradually add to the forest the minimally weighted edge that does not create a cycle.

Algorithm 46 Kruskal

```
function KRUSKAL($E, V, w$)
    Let $A$ be an empty set
    for $v \in V$
        MAKE-SET($v$)
    for $(u, v) \in E$ by ascending weight $w(u, v)$
        if FIND-SET($u$) != FIND-SET($v$) then
            $A = A \cup \{(u, v)\}$
            UNION($u, v$)
    return $A$
```

5.10 Bellman-Ford

BELLMAN-FORD(E, V, w, s) : $\Theta(E \cdot V)$

Input: A directed graph $G = (E, V)$ with weights $w : E \mapsto \mathbb{R}$ and a source vertex s

Output: The shortest path from s to all vertices in V

The idea is to keep track of the best estimate of the distance from s, $u.d$ for all vertices and its predecessor $u.p$ for all vertices and consider all edges V times and to decrease the minimum distance from the source if possible and update the predecessor. This method is guaranteed to succeed if there exists no negative weights cycles within the graph. This algorithm can hence be used to detect such cycles in a graph.

Algorithm 47 Relax

```
function RELAX($u, v, w$)        ▷ decreases the distance of $v$ if it is reachable more shortly from $u$
    if $v.d > u.d + w[u, v]$ then
        $v.d = u.d + w[u, v]$
        $v.p = u$
```

Algorithm 48 Bellman-Ford

```
function BELLMAN-FORD($E, V, w, s$)
    for $u$ in $V$
        $u.d = \infty$
        $u.p =$ NIL
    $s.d = 0$                                        ▷ Distances initialized
    for $i = 1$ to $V.size - 1$
        for $(u, v)$ in $E$
            RELAX($u, v, w$)
    for $(u, v)$ in $E$
        if $v.d > u.d + w[u, v]$ then
            return "Negative cycle present"
    return "Shortest paths done"
```

5.11 Dijkstra

DIJKSTRA(E, V, w, s) : $O(E \log V)$, even $O(V \log V + E)$

Input: A directed graph $G = (E, V)$ with positive weights $w : E \mapsto \mathbb{R}_+$ and a source vertex s

Output: The shortest path from s to all vertices in V

Dijkstra's shortest path algorithm is essentially a weighted BFS. Instead of enqueuing discovered vertices in a queue, we add them to a priority queue. At each step, we add the vertex closest to s.

Algorithm 49 Dijkstra

```
function DIJKSTRA($E, V, w, s$)
    for $u$ in $V$
        $v.d = \infty$
        $v.p =$ NIL
    $s.d = 0$                                        ▷ Distances initialized
    Let $S$ be an new empty set
    Let $Q$ be a mew min priority queue of vertices   ▷ The distance from the source is used as the key
    while $Q \neq$ NIL do
        $u =$ EXTRACT-MIN($Q$)
        $S = S \cup \{u\}$
        for $v$ in $Adj[u]$
            RELAX($u, v, w$)                          ▷ This also updates the queue with the new distances
```

5.12 All-pairs shortest path

Graph type	Algorithm	Time complexity	
unweighted	$\|V\|\times$ BFS	$O(VE)$	$\approx O(V^3)$
non-negative edge weights	$\|V\|\times$ DIJKSTRA	$O(VE + V^2 \log V)$	$\approx O(V^3)$
general	JOHNSON	$O(VE + V^2 \log V)$	$\approx O(V^3)$

Johnson's algorithm The idea is to reweight the graph with non negative weights (i.e. finding a function $h : V \mapsto \mathbb{R}$ such that $w_h(u, v) = w(u, v) + h(u) - h(v) \geq 0$) in order to use the $|V|\times$ DIJKSTRA method. With such a function, the distance between u and v, $\delta(u, v) = \delta_h(u, v) - h(u) + h(v)$

Procedure:

1:	BELLMAN-FORD from new vertex s	$O(VE)$
2:	Reweight all edges	$O(E)$
3:	$\|V\|\times$ DIJKSTRA	$O(VE + V^2 \log V)$
4:	Reweight all pairs	$O(V^2)$
Total :		$O(VE + V^2 \log V)$

Algorithm 50 Johnson

```
function JOHNSON($E, V, w$)
    Let $V' = V \cup s$
    Let $E' = E \cup \{(s, v) \mid v \in V\}$
    $w' = w \cup \{(s, v) \mapsto 0 \mid v \in V\}$    ▷ Extends the weight function to include the added edges
    if "Negative Cycle Present" == BELLMAN-FORD($E', V', w', s$) then
        return "Negative Cycle Present"
    Let $h[0 \dots V-1]$ be a new array
    for $v \in V$
        $h[v] = v.d$    ▷ Distance from $s$ to $v$ computed by BELLMAN-FORD with $w'$
    for $(u, v) \in E$
        $\hat{w}(u, v) = w(u, v) + h[u] - h[v]$
    Let $D[0 \dots V-1][0 \dots V-1]$ be a new array
    for $u \in V$
        DIJKSTRA($E, V, \hat{w}, u$)
        for $v \in V$
            $D[u][v] = v.d + h[v] - h[u]$    ▷ Distances from $u$ to $v$ computed by DIJKSTRA with $\hat{w}$
    return $D$
```

6 Dynamic Programming

The main Idea behind Dynamic Programming is to avoid doing calculations that were already carried out. Remembering previous results can save enormous amount of time.

The Problems that can be solved using Dynamic Programming are problems that exhibit an optimal substructure. i.e. it is possible to prove that a solution to the problem can be found amongst the solutions to problem generated by making an arbitrary choice to fix a parameter of the original problem. It is also necessary for a problem to exhibit several overlapping subproblems to have anything to gain from Dynamic programming.

Two approaches exist when designing a Dynamic Programming algorithm.

- **Top-down with memoisation:** Solve recursively but store each result in a table and check the table before computing anything
- **Bottom-up:** Sort the subproblems by size and solve the smaller subproblems first, that way, the bigger subproblems will never need to access the solution to a subproblem that has not been solved yet.

6.1 Fibonacci

Computing Fibonacci numbers is the easiest example of Dynamic Programming. A naive approach would run in exponential time whereas it is possible to find the solution in linear time.

NAIVE-FIB(n)	:	$T(n) = T(n-1) + T(n-2) = F(n) = \Theta((\frac{1+\sqrt{5}}{6})^n)$
TOP-DOWN-FIB(n)	:	$T(n) = \Theta(n)$
BOTTOM-UP-FIB(n)	:	$T(n) = \Theta(n)$

Algorithm 51 Naive-Fib

```
function NAIVE-FIB($n$)
    if $n \leq 1$ then
        return 1
    else
        return DUMB-FIB($n-1$) + DUMB-FIB($n-2$)
```

Here for the Memoized version, we initialize an array of size n with the value $-\infty$ and only actually compute a Fibonacci number when its corresponding entry in the array is $-\infty$ and we use the value in the array when it already exists.

Algorithm 52 Top-Down-Fib

```
function TOP-DOWN-FIB($n$)
    Let $r[0 \dots n]$ be a new array
    for $i = 0$ to n
        $r[i] = -\infty$
    return TOP-DOWN-FIB-AUX($n, r$)

function TOP-DOWN-FIB-AUX($n, r$)
    if $r[n] \neq -\infty$ then
        return $r[n]$
    else if $n = 0$ or $n = 1$ then
        return 1
    else
        $r[n] =$ TOP-DOWN-FIB-AUX($n - 1, r$) + TOP-DOWN-FIB-AUX($n - 2, r$)
        return $r[n]$
```

The bottom-up version of the dynamic programming algorithm makes its way up from the base case $n = 1$ or $n = 2$ until it reaches the desired number. Doing so it keeps the values it has already computed in an array. The advantage of the bottom-up version is that its runtime is way easier to derive than the top-down version.

Algorithm 53 Bottom-Up-Fib

```
function BOTTOM-UP-FIB($n$)
    Let $r[0 \dots n]$ be a new array
    $r[0] = 1$
    $r[1] = 1$
    for $i = 2$ to $n$
        $r[i] = r[i - 2] + r[i - 2]$
    return $r[n]$
```

6.2 Rod Cutting

Rod-Cut(n) : $T(n) = \Theta(n^2))$

The rod cutting-problem consists in finding the optimal cuts to perform in a rod to yield the maximum profit upon sale of the cut rods. It is assumed for simplicity that cuts can only be performed a multiple of a unit and that the price of sale is defined for each multiple those length. Here the substructure is that the optimal revenue is the price of sale of the first part of the rod after having performed the optimal first cut and the optimal revenue of the rest of the rod. This expression can be written as follows

$$r[n] = \begin{cases} 1 & n = p[1] \\ \max_{0 \leq j \leq n} \{p[j] + c[n - j]\} & \text{otw.} \end{cases}$$

Where $c[n]$ is the optimal revenue of a rod of size n, $p[n]$ is the price of sale of a rod of length n

The bottom-up algorithm for the rod cutting problem is th to progressively compute the optimal revenue of the rods from their smallest size to the one we need.

Algorithm 54 Rod-Cut

```
1: function Rod-Cut($p, n$)
2:     Let $r[0 \dots n]$ be a new array        ▷ $r[i]$ is the optimal revenue of a rod of length $i$
3:     Let $c[0 \dots n]$ be a new array        ▷ $c[i]$ is the optimal first cut to perform for a rod of length $i$
4:     $r[0] = 0$
5:     for $length = 1$ to $n$
6:         $max = -\infty$
7:         for $cut = 1$ to $length$
8:             if $p[cut] + c[length - cut] > max$ then
9:                 $c[length] = cut$
10:                $max = p[cut] + c[length - cut]$
11:        $r[length] = max$
    return $(r[n], c[n])$
```

6.3 Change making

Make-Change(n) : $T(n) = \Theta(n^2))$

Input: n distinct coins denominators $0 < w_1 < w_2 < \cdots < w_n$ and an amount which is are all positive integers

Output: The minimum number of coins necessary to make the change.

$$c[a] = \begin{cases} \infty & a < 0 \\ 0 & a = 0 \\ \min_{0<d<n} \{1 + c[a - w[d]]\} & \text{otw.} \end{cases}$$

Where $c[i]$ is the minimal number of coins necessary to make change on i, $w[i]$ is the denomination of the i-th coin.

Algorithm 55 Make-Change

```
1: function Make-Change($w, a$)
2:     Let $n[0 \dots n]$ be a new array        ▷ $r[a]$ is the optimal number of coins necessary for the amount $a$
3:     Let $c[0 \dots n]$ be a new array        ▷ $c[a]$ is the optimal first coin to give for an amount $a$
4:     $n[0] = 0$
5:     for $amount = 1$ to $a$
6:         $min = \infty$
7:         for $coin = 1$ to $n$
8:             if $1 + n[amount - w[coin]] < min$ then
9:                 $min = 1 + n[amount - w[coin]$
10:                $c[amount] = coin$
11:        $n[amount] = min$
    return $(n[a], c[a])$
```

6.4 Matrix chain multiplication

Make-Change(n) : $T(n) = \Theta(n^3))$

Input: A sequence of matrices $< A_1, A_2, \dots A_n >$, where for $i = 1, 2, \dots, n$, the matrix A has dimensions $p[i-1] \times p[i]$

Output: The full parenthesization of $\prod_{i=1}^{n} A_i$ in a way that minimizes the number of scalar multiplications.

The optimal substructure of this problem is that the optimal solution is the optimal first parenthesization and the parenthesization of both of its left and right terms.

$$m[s, e] = \begin{cases} 0 & s - e = 0 \\ min_{s<i<e} \{m[s, i] + m[i + 1, e] + p[i]\} & \text{otw.} \end{cases}$$

Where $m[s, e]$ is the minimal number of scalar multiplications carried out when computing $\prod_{i=s}^{e} A_i$

Algorithm 56 Parenthesize

```
function PARENTHESIZE($(p, n)$)
    Let $m[0 \dots n][0 \dots n]$ be a new array        ▷ $m[i, j]$ is the minimal number of scalar multiplications
    Let $c[0 \dots n][0 \dots n]$ be a new array        ▷ $c[i, j]$ is the optimal place to parenthesize
    for $i = 0$ to $n$
        $m[i][i] = 0$
    for $length = 1$ to $n$
        for $start = 0$ to $n - length$
            $end = start + length$
            $min = \infty$
            for $cut = start + 1$ to $end - 1$
                if $m[start][i] + p[i] + m[i+1][end]$ then
                    $min = m[start][i] + p[start-1]p[i]p[end] + m[i+1][end]$
                    $c[start][end] = i$
            $m[start][end] = min$
    return $(m[0][n], c[0][n])$
```

6.5 Longest common subsequence

LONGEST-COMMON-SUBSEQUENCE(S, T, n, m)	:	$T(n, m) = \Theta(n * m)$
PRINT-LCS(S, T, n, m)	:	$T(n, m) = \Theta(n * m) + \Theta(n + m)$

Input: Two sequences $S =< s_1, s_2, \dots, s_n >$ and $T =< t_1, t_2, \dots, t_n >$

Output: The longest common subsequence (not necessarily adjacent)

Algorithm 57 Longest-Common-Subsequence

```
function LONGEST-COMMON-SUBSEQUENCE($(S, T, n, m)$)
    Let $L[0 \dots n][0 \dots m]$ be a new array        ▷ The length of the longest common subsequence
    Let $M[0 \dots n][0 \dots m]$ be a new array        ▷ The sequence in which to search for LCS
    for $i = 1$ to $n$
        for $j = 1$ to $m$
            $L[i][j] = 0$
            $M[i][j] = null$
    for $i = 1$ to $n$
        for $j = 1$ to $m$
            if $S[i] == T[i]$ then
                $L[i][j] = L[i-1][j-1]$
                $M[i][j] =$"Both"
            else
                if $L[i-1][j] > L[i][j-1]$ then
                    $L[i][j] = L[i-1][j]$
                    $M[i][j] =$ "S"
                else
                    $L[i][j] = L[i][j-1]$
                    $M[i][j] =$ "T"
    return $(L, M)$
```

Algorithm 58 Print-LCS

```
function PRINT-LCS($S, T, n, m$)
    $(L, M)$ =LONGEST-COMMON-SUBSEQUENCE($S, T, n, m$)
    PRINT-HELPER($S, M, 0, 0, n$)
function PRINT-HELPER($S, M, i, j, n$)
    if $i == n$ then
        return
    else if $M[i][j]$ ="Both" then
        print $S[i]$
        PRINT-HELPER($S, M, i + 1, j + 1, n$)
    else if $M[i][j]$ ="S" then
        PRINT-HELPER($S, M, i + 1, j, n$)
    PRINT-HELPER($S, M, i, j + 1, n$)
```

6.6 Optimal Binary Search Tree

OPTIMIZE-BST(K, P, n) : $T(n) = \Theta(n^3)$

A binary Search Tree achieves searches in $O(h)$, where h is the height of the root. It can hence be both really efficient if it is optimally arranged or very badly otherwise. It is possible to minimise the expectation of the search time knowing the probability of the access to a certain key.

Input:

$K =< k_1, k_2, \ldots, k_n >$ an ordered sequence of keys (i.e. $i < j \Leftrightarrow k_i < k_j \quad \forall i, j$) ,

$P =< p_1, p_2, \ldots, p_n >$ the probability that the corresponding key is queried

Output: The optimal arrangement of the keys k into a BST with probabilities P

The expected time the search will take is

$$\begin{aligned}\mathbb{E} &= \sum_{i=1}^{n} (\text{depth}(k_i) + 1)\, p_i \\ &= 1 + \sum_{i=1}^{n} \text{depth}(k_i) p_i\end{aligned}$$

Algorithm 59 Optimize-BST

```
function OPTIMIZE-BST($K, P, n$)
    Let $E[1 \dots n][1 \dots n]$ be a new array    ▷ $E[i][j]$ The expected search time for BST from $i$ to $j$
    Let $R[1 \dots n][1 \dots n]$ be a new array    ▷ $R[i][j]$ The optimal root for BST from $i$ to $j$
    for $i = 1$ to $n$
        for $j = 1$ to $n$
            if $j < i$ then
                $E[i][j] = 0$
            else
                $E[i][j] = \infty$
    for $l = 1$ to $n$
        for $i = 1$ to $n - l$
            $j = i + l$
            $p = 0$
            for $k = i$ to $j$
                $p = p + P[k]$
            for $k = i$ to $j$
                if $E[i][j] > E[i][k] + E[k+1][j] + p$ then
                    $E[i][j] = E[i][k] + E[k+1][j] + p$
                    $R[i][j] = k$
    return $(E, R)$
```

6.7 Maximum Subarray