Algorithms and Data Structures*

October 1, 2014

Disclaimer

These notes have been prepared with the **only** purpose to help me pass the Computer Science qualifiying exam at the University of Illinois at Chicago. They are distributed as they are (including errors, typos, omissions, etc.) to help other students pass this exam (and possibly relieving them from part of the pain associated with such a process). I take **no responsibility** for the material contained in these notes (which means that you can't sue me if you don't pass the qual!) Moreover, this pdf version is distributed together with the original LaTeX (and LyX) sources hoping that someone else will improve and correct them. I mean in absolute no way to violate copyrights and/or take credit stealing the work of others. The ideas contained in these pages are **not mine** but I've just aggregated information scattered all over the internet.

Strongly suggested book

Before you even start reading this notes, do yourself a favor and go buy the "bible of algorithms" a.k.a. *Introduction to Algorithms, Third Edition* by Cormen, Laiserson, Rivest, Stein. (http://mitpress.mit.edu/algorithms/) It is very complete and clear. Also, thanks to the authors for the beautiful `clrscode3e` package that has been used to write the pseudocode of all the algorithms in these notes.

Contents

Important!!! Whenever talking about arrays we assume the index to ALWAYS start at 1 and NOT 0.

1 Analysis of algorithms[1]

- **Loop invariants** are useful to demonstrate the correctness of an algorithm. We need to show they hold during the three execution phases of an algorithm: *Initialization*, *Maintenance* and *Termination*. They work similarly to math induction.
- Goal is to compute the *running time* of an algorithm (on a particular input!).
- Various types of analysis are possible (which running time are we considering?): worst, best, average. These depend on which input we are considering.
  - Best case is rarely (never) used because it provides no useful information.
  - Average case is difficult to guess sometime.
  - Worst case gives an upper-bound for EVERY input, it occurs fairly often and often matches the average case.
- Instead of computing just the running time we are interested in the *order of growth* of the running time. (asymptotic analysis)

1.1 Asymptotic notation

Asymptotic notation applies to functions: we describe the running time of algorithms as a function of the input size n and we then apply the asymptotic notation to such a function.

Θ-notation For a given function $g(n)$, we denote $\Theta(g(n))$ the set of functions

$$\begin{aligned}\Theta(g(n)) &= \{(f(n): \text{ there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that} \\ &\quad 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}\end{aligned}$$

We write $f(n) = \Theta(g(n))$ to indicate $f(n) \in \Theta(g(n))$ and say that $g(n)$ is an *asymptotic tight bound* for $f(n)$.

O-notation For a given function $g(n)$, we denote $O(g(n))$ the set of functions

$$\begin{aligned}O(g(n)) &= \{(f(n): \text{ there exist positive constants } c \text{and } n_0 \text{ such that} \\ &\quad 0 \le f(n) \le cg(n) \, for \, all \, n \ge n_0\}\end{aligned}$$

We write $f(n) = O(g(n))$ to indicate $f(n) \in O(g(n))$ and say that $g(n)$ is an *asymptotic upper bound* for $f(n)$.

This notation is important because when we use O-notation to describe the worst-case running time of an algorithm we have a bound on the running time of the algorithm on *every input*. When we say "the running time of an algorithm is $O(g(n))$", we mean that there is a function $f(n)$ that is in $O(g(n))$ such that, no matter what particular input size n is chosen, the running time on that particular input is bounded from above by the value $f(n)$.

[1] Chapters 2, 3, 4

Ω-notation For a given function $g(n)$, we denote $\Omega(g(n))$ the set of functions

$$\begin{aligned}\Omega(g(n)) &= \{(f(n): \text{ there exist positive constants } c \text{and } n_0 \text{ such that} \\ &\quad 0 \leq cg(n) \leq f(n) \, for \, all \, n \geq n_0\}\end{aligned}$$

We write $f(n) = \Omega(g(n))$ to indicate $f(n) \in \Omega(g(n))$ and say that $g(n)$ is an *asymptotic lower bound* for $f(n)$.

When we say "the running time of an algorithm is $\Omega(g(n))$", we mean that no matter what particular input of size n is chosen for each value of n, the running time on that input is at least a constant times $g(n)$, for sufficiently large n. Equivalently we are giving a lower bound on the best-case running time of an algorithm.

1.1.1 Observations

- Note that in the definitions of all the previous asymptotic notations we assumed that every function is *asymptotically nonnegative* which means that all functions are nonnegative whenever n is sufficiently large.
- Note that it is important that we have some choices for the constants. Therefore if we find just one choice for the constants that is enough to apply the proper notation. Conversely if we need to show that the notation is not true we need to show that there is NO choice for the constants.
- In polynomials we can generally discard lower order terms when performing asymptotic analysis because they are insignificant for large n.
- For any two functions $f(n)$ and $g(n)$, we have

$$f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \, and \, f(n) = \Omega(g(n))$$

- Transitivity, reflexivity and symmetry in Θ-notation and transpose symmetry in Ω-notation and O-notation, apply to asymptotic comparison.

o-notation For a given function $g(n)$, we denote $o(g(n))$ the set of functions

$$\begin{aligned}o(g(n)) &= \{(f(n) : \forall c > 0, \exists n_0 > 0 : \\ &\quad 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}\end{aligned}$$

We write $f(n) = o(g(n))$ to indicate $f(n) \in o(g(n))$ and say that $f(n)$ is *asymptotically smaller* than $g(n)$.

Intuitively the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity; that is,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

ω-notation For a given function $g(n)$, we denote $\omega(g(n))$ the set of functions

$$\begin{aligned}\omega(g(n)) &= \{(f(n) : \forall c > 0, \exists n_0 > 0 : \\ &\quad 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}\end{aligned}$$

We write $f(n) = \omega(g(n))$ to indicate $f(n) \in \omega(g(n))$ and say that $f(n)$ is *asymptotically larger* than $g(n)$.
We can also write $f(n) = \omega(g(n)) \iff g(n) = o(f(n))$.
The relation $f(n) = \omega(g(n))$ implies that

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

if the limit exists.

1.1.2 Amortized analysis

Amortized analysis considers the entire sequence of operations of the program. It is based on the idea that, while certain operations may be extremely costly in resources, they cannot occur at a high enough frequency to weigh down the entire program because the number of less costly operations will far outnumber the costly ones in the long run, "paying back" the program over a number of iterations.

1.2 Common functions (refresher)

1.2.1 Monotonicity

A function $f(n)$ is *monotonically increasing* if $m \leq n$ implies $f(m) \leq f(n)$. Similarly, it is *monotonically decreasing* if $m \leq n$ implies $f(m) \geq f(n)$.

1.2.2 Floors and ceilings

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$$

1.2.3 Exponentials

- $a^{-1} = \frac{1}{a}$
- For all n and all $a \geq 1$ the function a^n is monotonically increasing in n.
- For all real constants $a > 1$ and b, $n^b = o(a^n)$.

1.2.4 Logarithms

- Remember that $\log_a b = c$ means $a^c = b$.
- When we write $\log n$ we mean $\log_2 n$.
- $\ln n = \log_e n$
- $a = b^{\log_b a}$
- $a^{\log_b n} = n^{\log_b a}$
- $\log_b a = \frac{\log_c a}{\log_c b}$
- $\log_b a = \frac{1}{lob_a b}$
- If the base is strictly greater than one then $\log_b a$ is strictly increasing.

Note how:

$$log_b a \begin{cases} > 1 & \text{if } a > b \\ = 1 & \text{if } a = b \\ < 1 & \text{if } a < b \end{cases}$$

We say that a function $f(n)$ is polylogarithmically bounded if $f(n) = O(\lg^k n)$ for some constant k. For any constant $a > 0$, $\lg^b n = o(n^a)$.

1.2.5 Iterated logarithm

The iterated logarithm is a very slowly growing function that it is defined as

$$\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}$$

that is the smallest number of times we have to iteratively apply the function lg to n in order to obtain a value smaller or equal than 1.

1.2.6 Factorials

A weak upperbound on the factorial function is $n! \leq n^n$, since each of the n terms in the factorial product is at most n,

Stirling's approximation:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$$

1.2.7 Ackermann function

For integers $k \geq 0$ and $j \geq 1$, we define the function $A_k(j)$ as

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1 \end{cases}$$

where the expression $A_{k-1}^{(j+1)}(j)$ uses the same functional iteration notation of the iterated logarithms. Specifically, $A_{k-1}^{(0)}(j) = j$ and $A_{k-1}^{(i)}(j) = A_k\left(A_{k-1}^{(i-1)}(j)\right)$ for $i \geq 1$. We refer to the parameter k as the level of the function A.

Ackermann function value grows rapidly, even for small inputs.

1.2.8 Inverse Ackermann function

We define the inverse of the function $A_k(n)$, for integer $n \geq 0$, by

$$\alpha(n) = \min\{k : A_k(1) \geq n\}$$

In words, $\alpha(n)$ is the lowest level k for which $A_k(1)$ is at least n. We can see that

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2 \\ 1 & \text{for } n = 3 \\ 2 & \text{for } 4 \leq n \leq 7 \\ 3 & \text{for } 8 \leq n \leq 2027 \\ 4 & \text{for } 2027 \leq n \leq A_4(1) \end{cases}$$

which means that $\alpha(n) \leq 4$ for all practical purposes. (which means it can be treated as a constant)

1.3 Recursive (Divide-and-Conquer) algorithms analysis

In Divide-and-Conquer algorithms, we solve a problem recursively, applying three steps at each level of the recursion:

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough just solve the subproblems in a straightforward way.

Combine the solutions to the subproblems into the solution for the original problem.

1.3.1 Master method (Master Theorem of recursion)

Provides a "canned" method for solving recurrences of the form $T(n) = aT(n/b) + f(n)$.

Theorem Let $a, b \in \mathbb{N} : a \geq 1, b > 1$ be constant and $f(n)$ be a function and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n)$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

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

Meaning Intuitively, we can see a as the number of subproblems, n/b as the size of each subproblem and $f(n)$ as the time needed to divide and combine the results of the subproblems.

In each of the three cases, we compare the function $f(n)$ to the function $n^{\log_b a}$: intuitively, the larger of the two functions determines the solution to the recurrence.

1. In this case $n^{\log_b a}$ is the largest, therefore $T(n) = \Theta(n^{\log_b a})$.
2. In this case functions are comparable, therefore $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(f(n) \log n)$.
3. In this case$f(n)$ is the largest, therefore $T(n) = \Theta(f(n))$.

Note that, in order for the master theorem to hold, $f(n)$ needs to be polynomially smaller (bigger) than $n^{\log_b a}$. That is, $f(n)$ must be asymptotically smaller (bigger) by a factor n^ϵ for some constant $\epsilon > 0$.

Simplified version Assuming $f(n) = cn^k$ and $T(1) = c$ we obtain:

$$T(n) = aT(n/b) + cn^k = \begin{cases} \Theta(n^{\log_a b}) & if\, a > b^k \\ \Theta(n^k \log n) & if\, a = b^k \\ \Theta(n^k) & if\, a < b^k \end{cases}$$

where k is the "order" of the combination and we can see $n = b^m$ where m are the steps in the recursion. We can obtain a very particular case when $k = 1$ that is, the time required to combine the results is linear.

1.3.2 Recursion-tree method

When the master theorem can't be applied, we apply the recursion-tree method to generate a good guess for a solution to a recurrence which can then be verified using induction. The process can be described with the following steps:

1. Convert the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. Leaves are the subproblems while the cost for dividing and combining is annotated on the side of the tree.
2. Find a closed formula to describe the problem size based on the level of the recursion-tree i.
3. Calculate the height of the tree k using (2) and knowing that, at the deepest level, the problem size is 1 (or 2, or whatever else).
4. Find a closed formula to describe the number of subproblems at each level of the tree based on the level of the recursion-tree i.
5. Find a closed formula to describe the cost for dividing and combining at each level of the tree based on the level of the recursion-tree i.
6. Find the cost at each level of the tree by multiplying (4) by (5).

7. Calculate the cost at the deepest level k multiplying (4) by $T(1)$, assuming each problem at the deepest level costs $T(1)$.

8. Calculate the entire recursion-tree cost by adding up the costs over all levels.

9. Find a closed form for the summation obtained in (8).

1.3.3 Substitution method

In the substitution method, we guess a bound (by "expanding" the recursion) and then use mathematical induction to prove our guess correct.

2 Linear data structures[2]

2.1 Lists

A *list* is simply a finite and *ordered* sequence of elements.

Operation	Cost
Access (indexing)	$O(1)$
Insert / Delete at end	$O(1)$
Insert / Delete at i	$O(n-i)$

Table 1: Operations cost in array based implementation

Note that whenever we use an array implementation for lists we need to know the maximum size of the list a priori, unless we use `ArrayLists` (i.e. dynamic arrays) which combine benefits of the operation costs of arrays (although amortized) with flexible size of linked lists. ArrayLists implementations typically double in size when full.

Note that access (indexing) takes constant time because we assume that the index of the element we are trying to access is known.

Note that every time we insert/delete a value from a sorted *array* we need to shift all the other elements in order to restore the order; which is why insert and delete take $O(n)$.

Operation	Cost
Access (indexing)	$O(i)$
Insert / Delete at end	$O(1)$
Insert / Delete at i	$O(1)$

Table 2: Operations cost in linked list based implementation

Note that all operations have running time $O(1)$ because we assume that the pointer to the previous element is known (doubled linked list): therefore the only operation that needs to be performed is the update of the pointers.

2.1.1 Linked-lists in Java

A note on `Vector`: it is very similar to ArrayList, but it's thread safe and its size is incremented in chunks.

[2] Chapter 10

```
class Node {
  int val;
  Node next = null;

  Node(int v) {
    val = v;
  }
}
```

2.2 Stacks

Stacks are a restricted variant of a list in which elements can be inserted and removed from only one end. (Last In First Out - LIFO) Insertion is called *push*, deletion is called *pop* and access is called *peek*. (i.e. check which element is on top of the stack)

Operation	Cost
Peek	$O(1)$
Push	$O(1)$
Pop	$O(1)$

Table 3: Cost of operations (both array and linked list based implementation)

2.3 Queues

Restricted variant of a list in which elements can only be inserted at the back and deleted from the front. (First In First Out - FIFO) Insertion is called *enqueue*, deletion is called *dequeue* and access is called *peek*. (i.e. check which element is on front of the stack)

Operation	Cost
Peek	$O(1)$
Enqueue	$O(1)$
Dequeue	$O(1)$

Table 4: Cost of operations (both array and linked list based implementations)

Note that to obtain $O(1)$ for all operations in the array implementation we need to use *circular arrays*.

Note that to obtain $O(1)$ for all operations in the linked list implementation we need to maintain two pointers, one to the head and one to the tail of the list.

3 Hash Tables[3]

A hash table generalizes the simpler notion of an ordinary array. Directly addressing into an ordinary array makes effective use of our ability to examine an arbitrary position in an array in $O(1)$ time. Hash tables typically use an array of size proportional to the number of keys actually stored. Instead of using the key as an array index directly, the array index is computed from the key using a *hash function*. We can deal with collisions using chaining. Storage can be reduced to $\Theta(n)$, where is n is the cardinality of the keys set of keys K. Searching for an element is $O(1)$ on average (compared to $O(1)$ worst case for direct addressing). Worst case time serach for Hash Table is $O(n)$ when all the keys collide and are stored in a single linked list.

[3]Chapter 11

4 Binary trees[4]

Two implementations are possible.

Linked list based: Each node contains some satellite data, the key and two pointers to the left and right subtree respectively. There might be also a third pointer to the parent node.

Array based: Efficient for complete binary trees because they have only one possible shape. The root of the tree is $A[1]$ and we need to implement:

`Parent(i) {return ⌊i/2⌋}` returns the parent,

`Left(i) {return 2*i}` returns the left subtree,

`Right(i) {return 2*i+1}` returns the right subtree.

4.1 Binary heaps

A *binary heap* is a complete binary tree whose values are partially ordered according to the *heap property*. There are two kinds of heaps:

max-heap: $A[i] \leq Parent(A[i])$ which means the *largest* element is stored at the root. This kind of heap is used in the *heap sort* algorithm.

min-heap: $A[i] \leq Parent(A[i])$ which means the *smallest* element is stored at the root. This kind of heap is usually used to implement *priority queue*.

- Binary heaps are efficiently implemented using arrays (since they are complete binary trees)
- There is NO relation between a node and its siblings (hence the partial order).

Operation	Cost
Peek root (max/min)	$O(1)$
Insert	$O(\log n)$
Delete	$O(\log n)$
Build	$O(n)$

Table 5: Cost of operations

Note that since root access can be done in $O(1)$, binary heaps are very efficient every time we need to compute the maximum or minimum.

4.1.1 Maintaining the heap property

We consider a max-heap but the same results are true for the min-heaps.

Given a max-heap A and an index i the following procedure checks if the max-heap property is maintained for $A[i]$ and, if not, lets the value $A[i]$ *sift-down* in the max-heap so that the subtree rooted at index i obeys the max-heap property.

[4]Chapters 6, 12, 13, 21

MAX-HEAPIFY(A, i)

```
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4        largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7        largest = r
8  if largest ≠ i
9        exchange A[i] with A.largest
10       MAX-HEAPIFY(A, largest)
```

Running time of this procedure is $O(\log n)$ (if the recursive call doesn't happen it is obviously $O(1)$).

MAX-HEAPIFY is used as a subprocedure in BUILD-MAX-HEAP which builds a heap from an unordered array in $O(n)$.

4.1.2 Priority queues

A *priority queue* is a data structure for maintaining a set S of elements, each with an associated value k called key (which represents also the priority). As in queues we remove elements from the head but, whenever we need to insert a new element in a priority queue, we can't just enqueue it at the tail but we need to insert it in the proper place according to its priority.

Enqueue After we insert a new value at the end of the heap we use the following procedure to *sift-up* the newly inserted value so that the max-heap property is restored.

MAX-HEAP-INSERT(A, key)

```
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = key
3  i = A.heap-size
4  while i > 1 and A[PARENT(i)] < A[i]
5        exchange A[i] with A[PARENT(i)]
6        i = PARENT(i)
```

Running time of this procedure is $O(\log n)$.

Dequeue We return $A[1]$, store $A[A.heap\text{-}size]$ in $A[1]$, decrease $A.heap\text{-}size$ and run MAX-HEAPIFY$(A, 1)$ to restore the max-heap property.

Operation	Cost
Peek	$O(1)$
Enqueue	$O(\log n)$
Dequeue	$O(\log n)$
Build	$O(n)$

Table 6: Cost of operations

Note how cost of enqueue is $O(\log n)$ due to the fact that we need to maintain the heap property after we insert a new element in the tree.

Note how cost of dequeue is $O(\log n)$ due to the fact that, even if it cost only $O(1)$ to access the root of the tree, we need to maintain the heap property since we are removing the root itself.

4.2 BSTs

A *binary search tree* (BST) is a particular type of binary tree where the keys are always stored in such a way as to satisfy the *binary-search-tree property*.

Let x be a node in a BST. If y is a node in the left subtree of x then $y.key \leq x.key$. If y is a node in the right subtree of x then $y.key \geq x.key$.

- Note how this is different from just asking the left child is smaller and the right child is bigger than the parent.
- BSTs are usually implemented using linked lists since they don't have any shape restriction.

4.2.1 Searching

Given a key k and a BST x we want to know if x contains k.

```
BINARY-SEARCH(x, k)
1  if x == NIL or k == x.key
2       return x
3  If k < x.key
4       return BINARY-SEARCH(x.left, k)
5  else return BINARY-SEARCH(x.right, k)
```

This is the BINARY-SEARCH algorithm, a very prominent example of Divide-and-Conquer approach, whose running time is $O(h)$.

4.2.2 Inserting

To insert a new value v into a BST T, we use the procedure TREE-INSERT. The procedure takes a node z for which $z.key = v$ and modifies T and some of the attributes of z such that it inserts z into the appropriate position in the tree.

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL              // find right place
 4       y = x
 5       if z.key < x.key
 6            x = x.left
 7       else x = x.right
 8  z.p = y                    // connect z to the rest of the tree
 9  if y == NIL
10       T.root = z            // tree is empty
11  elseif z.key < y.key
12       y.left = z
13  else y.right = z
```

4.2.3 Deleting

The overall strategy for deleting a node z from a BST T has three basic cases but one of these cases is a bit tricky.

- If z has no children, then we simply remove it by modifying its parent to replace z with NIL as its child.
- If z has just one child, then we elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child.

- If z has two children, then we find z's successor y which is contained in z's right subtree (it is the smallest element greater than z) and have y take z's position in the tree. The rest of z's original right subtree becomes y's new right subtree, and z's left subtree becomes y's new left subtree.

Operation	Cost
Search	$O(h)$
Insert/Delete	$O(h)$
Max/Min	$O(h)$
Prec./Succ.	$O(h)$

Table 7: Cost of operations

Note how all the basic operations on a BST take time proportional to the height h of the tree.

4.2.4 Other operations

BSTs also support other operations including MINIMUM, MAXIMUM, PREDECESSOR and SUCCESSOR, all of which run in $O(h)$.

Note how the BST property allows us to print out all the keys in a BST in sorted order in $O(n)$ time just performing an inorder visit.

4.3 Self-balancing BSTs

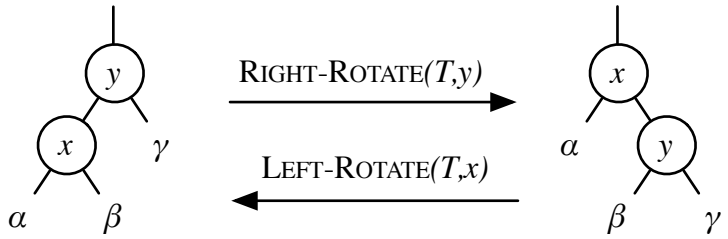
Since all basic BST operations execute in $O(h)$ they are efficient if h is small. However, if the tree is unbalanced, h can become as large as n (i.e. a simple path) causing all BST operations to execute in $O(n)$, which is no faster than using a linked list! Self-balancing BSTs try to keep the tree height minimal, by performing transformations (such as rotations) at key times on the tree, so that all the basic BST operations take $O(\log n)$.

Operation	Cost
Search	$O(\log n)$
Insert/Delete	$O(\log n)$
Max/Min	$O(\log n)$
Prec./Succ.	$O(\log n)$

Table 8: Cost of operations

4.3.1 Rotations

Binary tree rotations are operations on a binary tree that change the structure (i.e. the pointers) without interfering with the order of the elements (which means the binary-search-tree property is maintained). A tree rotation moves one node up in the tree and one node down. The idea is to decrease the height of the tree by moving smaller subtrees down and larger subtrees up. There are two kind of rotations, left and right, which are showed in the following picture.



The pseudocode for LEFT-ROTATE assumes that $x.right \neq T.nil$

```
LEFT-ROTATE(T,x)
 1  y = x.right              // set y (extract it from x)
 2  x.right = y.left         // save y's left subtree into x's right subtree
 3  if x.right ≠ T.nil
 4      y.left.p = x
 5  y.p = x.p                // link x's parent to y
 6  if x.p == T.nil
 7      T.root = y
 8  elseif x == x.p.left
 9      x.p.left = y
10  else x.p.right = y
11  y.left = x               // put x on y's left
12  x.p = y
```

Symmetrically the pseudocode for RIGHT-ROTATE assumes that $y.left \neq T.nil$. We can easily obtain it from LEFT-ROTATE substituting x with y and $right$ with $left$.

4.3.2 AVL Trees

An *AVL-tree* is a binary tree where the heights of the two child subtrees of any node differ by at most one; therefore, it is also said to be height-balanced.

The *balance factor* of a node is the height of its left subtree minus the height of its right subtree. A node with balance factor 1, 0, or −1 is considered balanced while a node with any other balance factor is considered unbalanced and requires rebalancing the tree.

Insertion The pseudocode for AVL-INSERT assumes that every node x in the AVL-tree stores its balance factor bf. Every leaf and newly inserted node z have $bf = 0$.

```
AVL-INSERT(T, z)
 1  TREEINSERT(T, z)
 2  if z.p.left == z                          // changes balance factor in z's parent
 3      z.p.bf = z.p.bf + 1
 4  else z.p.bf = z.p.bf − 1
 5  x = z.p
 6  while x ≠ NIL and x.bf ≠ 0 and |x.bf| < 2  // propagates balance factor changes
 7      if x.p.left == x
 8          x.p.bf = x.p.bf + 1
 9      else x.p.bf = x.p.bf − 1
10      x = x.p
11  if x.bf == −2                             // x's right subtree is unbalanced
12      if x.right.bf == −1
13          LEFT-ROTATE(x)
14      else RIGHT-ROTATE(x.right)
15          LEFT-ROTATE(x)
16  if x.bf == 2                              // x's left subtree is unbalanced
17      if x.left.bf == 1
18          RIGHT-ROTATE(x)
19      else LEFT-ROTATE(x.left)
20          RIGHT-ROTATE(x)
```

Deletion The overall strategy for deleting a node z from an AVL tree T has the same three basic cases we have already seen in BSTs.

- If z has no children, then we simply remove it by modifying its parent to replace z with NIL as its child.
- If z has just one child, then we elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child.
- If z has two children, then we find z's successor y which is contained in z's right subtree (it is the smallest element greater than z) and have y take z's position in the tree. The rest of z's original right subtree becomes y's new right subtree, and z's left subtree becomes y's new left subtree. After deletion, retrace the path back up the tree from y's parent to the root, adjusting the balance factors as needed. The retracing can stop if the balance factor becomes -1 or $+1$ indicating that the height of that subtree has remained unchanged. If the balance factor becomes 0 then the height of the subtree has decreased by one and the retracing needs to continue. If the balance factor becomes -2 or $+2$ then the subtree is unbalanced and needs to be rotated to fix it. If the rotation leaves the subtree's balance factor at 0 then the retracing towards the root must continue since the height of this subtree has decreased by one. This is in contrast to an insertion where a rotation resulting in a balance factor of 0 indicated that the subtree's height has remained unchanged.

4.3.3 Red-Black trees

A *red-black tree* is a binary tree that satisfies the following red-black properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node x, all simple paths form x to descendant leaves contain the same number of black nodes. We call such a number the *back-height* of x, denoted $\text{bh}(x)$. The black-height of a tree is the black-height of its root.

Note how the leaves here are simply the sentinel value $T.nil$ which is used to treat a NIL child of a node x as an ordinary node whose parent is x.

A red black tree with n internal nodes has height at most $2\log(n+1)$.

4.4 Tries (or Radix trees)

A trie is a variant of an n-ary tree in which characters are stored at each node. Each path down the tree may represent a word. Radix trees are often called "tries," which comes from the middle letters in the word retrieval. It is good whenever the alphabed is limited and there is a high degree of redundancy in the first portion of the word (sequences share prefixes).

4.5 Union-Find

A *disjoint-set data structure* (or *Union-Find data structure*) maintains a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets and, for each element $x \in S_i$, $1 \leq i \leq k$ supports the following operations:

- MAKE-SET(x) which creates a new set whose only member is x. Since sets are disjoint, we require that x not already be in some other set.
- UNION(x, y) which unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is $S_x \cup S_y$.
- FIND-SET(x) which returns the dynamic set that contains x.

Usually each set is referenced by a representative, that is an element x of such a set and we represent the overall number of elements x in the data structure with n. Also, we assume that the n MAKE-SET operations are the first n operations performed when building a new disjoint-set data structure.

Union-Find data structures are used to represent the connected components of an undirected graph and in particular in Kruskal's Algorithm to find the Minimum Spanning Tree in an undirected graph.

4.5.1 Implementation

Two implementations are possible:

linked-lists: each set is represented by its own linked list. With this implementation, both MAKE-SET and FIND-SET are easy, and require $O(1)$ time. To carry out MAKE-SET(x), we create a new linked-list whose only object is x. For FIND-SET(x), we just follow the pointer from x back to its set object and return it. However UNION(x, y) procedure requires an average of $\Theta(n)$ time per call because, after appending y to x, we must update the pointers of all elements in y's list.

forest: each set is represented by a rooted tree. AMAKE-SET operation simply creates a tree with just one node. We perform a FIND-SET operation by simply following parent pointers until we find the root of the tree. A UNION operation causes the root of one tree to point to the root of the other.

Operation	Cost
Make-Set	$O(1)$
Union	$O(\alpha(n))$
Find-Set	$O(\alpha(n))$

Table 9: Cost of operations

Note that these values refer to the optimal forest implementation.

There is a nice theorem that states that a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, can be performed on a disjoint-set forest in worst-case time $O(m\alpha(n))$. Note that, since each UNION operation reduces the number of sets by one, the maximum number of such operations is $n - 1$. Also note that $m \geq n$ and that we assume that the first n MAKE-SET operations are the first n operations performed.

5 Sorting, selecting, searching[5]

5.1 Insertion sort

The *insertion sort* algorithm is the most intuitive, yet inefficient, sorting algorithm. It orders the sequence by inserting each element in the right place in the sorted portion of the sequence.

5.1.1 Implementation

The following procedure takes as a parameter an array $A[1\,..\,n]$ containing a sequence of numbers of length n and sorts it *in place* by rearranging the numbers within the array itself. This means the input array A contains the sorted output sequence when the procedure is finished.

[5]Chapters 2, 6, 7, 8, 9

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j-1].
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i+1] = A[i]
7          i = i - 1
8      A[i+1] = key
```

5.1.2 Time complexity

$O(n^2)$: When the array is reverse sorted order, which is the worst case, we need to compare each element $A[j]$ with each element in the entire sorted subarray $A[1..j-1]$.

5.2 Merge sort

The *merge sort* algorithm closely follows the Divide-and-Conquer paradigm. Intuitively, it operates as follows:

Divide: Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted subsequences to produce the sorted answer.

Merge sort does not sort in place but it is possible to devise a modified version that does so.

5.2.1 Implementation

The following procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are already sorted and it merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$.

MERGE(A, p, q, r)

```
 1  n1 = q - p + 1
 2  n2 = r - q
 3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i - 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

We can use the MERGE procedure as a subroutine in the merge sort algorithm. The following procedure sorts the elements in the subarray $A[p..r]$.

MERGE-SORT(A, p, r)

```
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

To sort the entire array A we make the initial call MERGE-SORT$(A, 1, A.length)$.

5.2.2 Time complexity

$\Theta(n \log n)$, since the MERGE procedure, which runs in $\Theta(n)$, is executed $\log n$ times.

5.3 Heap sort

The *heap sort* algorithm works by exploiting the max-heap property that the biggest element is always at the root. It swaps the root of the heap with the last element in the array and then reorganizes all the elements but the last one so that the data structure satisfies again the max-heap property.

5.3.1 Implementation

The following procedure takes as a parameter an array $A[1 \,..\, n]$ and sorts it *in place*. The procedure BUILD-MAX-HEAP(A) is not described in this document.

HEAP-SORT(A)

```
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

5.3.2 Time complexity

$\Theta(n \log n)$, since the call to BUILD-MAX-HEAP takes time $O(n)$ and each of the $n-1$ calls to MAX-HEAPIFY takes time $O(\log n)$.

5.4 Quick sort

The *quick sort* algorithm is another example of application of the Divide-and-Conquer paradigm. However, differently from merge sort, it sorts *in place*. The three steps to sort a subarray $A[p \,..\, r]$ are:

Divide: Partition (rearrange) the array $A[p \,..\, r]$ into (possibly empty) subarrays $A[p \,..\, q-1]$ and $A[q+1 \,..\, r]$ such that each element of $A[p \,..\, q-1]$ is less or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1 \,..\, r]$. Compute the index q as part of this partition procedure.

Conquer: Sort the two subarrays $A[p \,..\, q-1]$ and $A[q+1 \,..\, r]$ by recursive calls to quick sort.

Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p \,..\, r]$ is now sorted.

5.4.1 Implementation

The following procedure implements quick sort.

QUICK-SORT(A, p, r)

```
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICK-SORT(A, p, q − 1)
4      QUICK-SORT(A, q + 1, r)
```

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p\,..\,r]$ in place by partitioning the array in three (possibly empty) regions: elements $\leq x$, elements $> x$ and x.

PARTITION(A, p, r)

```
1  x = A[r]                          // selects the pivot
2  i = p − 1
3  for j = p to r − 1                // separates the elements ≤ x from the ones > x
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]       // moves the pivot "in the middle"
8  return i + 1
```

To sort the entire array A we make the initial call QUICK-SORT$(A, 1, A.\mathit{length})$.

5.4.2 Time complexity

The running time of quick sort depends on whether the partition is balanced ($\Theta(n \log n)$) or unbalanced ($\Theta(n^2)$). The average-case running time of quick sort is $O(n \log n)$ which is much closer to the best case than to the worst case.

It is possible to devise a randomized version of quick sort where the pivot is chosen randomly among the elements of the subarray $A[p\,..\,r]$ yielding an average running time of $O(n \log n)$.

5.5 Lower bounds for sorting

All the previous sorting algorithms belong to the family of the so called *comparison sorts*. This family is important because we can prove that any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

This means that heap sort and merge sort are *asymptotically optimal* comparison sorts.

5.6 Counting sort, radix sort and bucket sort

These sorting algorithms have an average case of $O(n)$. They are faster than comparison sort algorithms because they assume something about the input.

Counting sort assumes that each of the n input elements is an *integer* in the range *0* to k, for some integer k. When $k = O(n)$, the sort runs in $\Theta(n)$ time. Counting sort determines, for each input element x, the number of elements less than x. It uses this information to place element x directly into its position in the output array. Care must be placed to handle duplicate entries. In the code for counting sort, we assume that the input is an array $A[1\,..\,n]$, and thus $A.\mathit{length} = n$. We require two other arrays: the array $B[1\,..\,n]$ holds the sorted output, and the array $C[0\,..\,k]$ provides temporary working storage.

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j =A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

Bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval $[0, 1)$. Bucket sort divides the interval $[0, 1)$ into n equal-sized subintervals, or buckets, and then distributes the n input numbers into the buckets. Since the inputs are uniformly and independently distributed over $[0, 1)$, we do not expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each. Code for bucket sort assumes that the input is an n-element array A and that each element $A[i]$ in the array satisfies $0 \leq A[i] < 1$. The code requires an auxiliary array $B[0..1]$ of linked lists (buckets) and assumes that there is a mechanism for maintaining such lists.

```
BUCKET-SORT(A)
1  let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4      make B[i] an empty list
5  for i = 1 to n
6      inser A[i] into list B[⌊nA[i]⌋]
7  for i = 0 to n − 1
8      sort list B[i] with INSERTION-SORT
9  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

5.7 Binary search

See page 11.

5.8 Quick select

A *selection algorithm* is an algorithm for finding the ith smallest number in a set of n (distinct) elements. Such a number is called the ith order statistic of the set. This includes the cases of finding the minimum (first order statistics), maximum (nth order statistics), and median ($\lfloor (n+1)/2 \rfloor$ lower median or $\lceil (n+1)/2 \rceil$ upper median) elements.

We can solve the selection problem in $O(n \log n)$ time, since we can sort the numbers and then simply access the ith element in the output array. However, there are $\Theta(n)$ time algorithms to determine the minimum and the maximum and the following algorithm, to solve the general selection problem, is faster.

5.8.1 Implementation

The following procedure implements the quick select algorithm which works similarly to quick sort: it takes as input the array $A[p..r]$ and returns its ith smallest element.

RANDOMIZED-SELECT(A, p, r, i)

```
1  if p == r
2      return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k                                   // the pivot is the answer
6      return A[q]
7  elseif i < k
8  return RANDOMIZED-SELECT(A, p, q − 1, i)    // continue search within the smaller elements
9  else return RANDOMIZED-SELECT(A, q + 1, r, i)   // continue search within the bigger elements
```

The algorithm uses the RANDOMIZED-PARTITION procedure, which is the randomized version of the PARTITION procedure used by quick sort.

RANDOMIZED-PARTITION(A, p, r)

```
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

5.8.2 Time complexity

The expected running time of RANDOMIZED-SELECT is $\Theta(n)$. This is due to the fact that, as in quick sort, we partition the input array recursively but unlike quick sort, which recursively processes both sides of the partition, RANDOMIZED-SELECT works only on one side of the partition. (the one containing the i-th element)

6 Patterns in algorithms design[6]

6.1 Dynamic Programming

An *optimization problem* needs to have the following two characteristics in order for DP to apply:

- expose *optimal substructure*, which means that an optimal solution to a problem contains optimal solutions to smaller subproblems (exactly like in Divide-and-Conquer)
- have *overlapping problems*, which means that a subproblem is revisited multiple times. If problems are not overlapping it doesn't make sense to store the partial results and a Divide-and-Conquer algorithm is probably a better choice.

DP algorithms work in a bottom-up fashion by finding optimal solutions to subproblems (often all of them) and then making a choice among these subproblems to select which one will be used in solving the bigger subproblems and ultimately the original problem.

6.1.1 Longest Common Subsequence (LCS)

Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a subsequence of X if there exists a strictlyr increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$. Given two sequences X and Y, we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y.

In the longest subsequence problem, we are given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ and wish to find a maximum length common subsequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ of X and Y.

[6] Chapters 15, 16

Optimal substructure and overlapping problems In order to apply DP we need to show that the problem has optimal substructure. This step will also offer a hint on how to use the subproblems solution to find the solution to the problem.

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

We can also see how to find an LCS of X and Y, we may need to find the LCS of X and Y_{n-1} and of X_{m-1} and Y.

Implementation We can define $c[i, j]$ to be the length of an LCS of the sequences X_i and Y_j. The optimal substructure of the LCS problem gives the recursive formula

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{and } z_i = y_j \\ max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{and } z_i \neq y_j \end{cases}$$

We can use such a recursion to write an algorithm that takes in input X and Y and stores the values of $c[i, j]$ in a table $c[0\,..\,m, 0\,..\,n]$, and it computes the entries in row-major order. The procedure also maintains a table $b[1\,..\,m, 1\,..\,n]$ to help us construct an optimal solution: intuitively, $b[i, j]$ points to the table entry corresponding to the optimal subproblem solution chosen when computing $c[i, j]$. The procedure returns tables b and c and the LCS for X and Y is stored in $c[m, n]$.

LCS-LENGTH(X,Y)

```
 1  m = X.length
 2  n = Y.length
 3  let c[0..m, 0..n] and b[1..m, 1..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = " ↖ "
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = " ↑ "
16          else c[i, j] = c[i, j − 1] + 1
17              b[i, j] = " ← "
18  return c and b
```

We can then print the LCS using table b and the following procedure. The initial call is PRINT-LCS(b,X,$X.length$,$Y.length$).

```
PRINT-LCS(b,X,i,j)
1  if i == 0 or j == 0
2       return
3  if b[i, j] == " ↖ "
4       PRINT-LCS(b,X,i − 1,j − 1)
5       print x_i
6  elseif b[i, j] == " ↑ "
7       PRINT-LCS(b,X,i − 1,j)
8  else PRINT-LCS(b,X,i,j − 1)
```

6.1.2 Other problems that are solvable with DP algorithms

Rod cutting problem Given a rod of length n and a table of prices p_i for $i = 1, 2 \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

The following dynamic programming algorithm takes as input an array $p[1\,..\,n]$ of prices and an integer n and returns an array $r[0\,..\,n]$ in which it saves the results of the subproblems and an array $s[0\,..\,n]$ containing the optimal size of the first piece to cut off when solving a subproblem of size j.

```
CUT-ROD(p, n)
 1  let r[0..n] and s[0..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4       q = −∞
 5       for i = 1 to j
 6            if q < p[i] + r[j − i]
 7                 q = p[i] + r[j − i]
 8                 s[j] = i
 9       r[j] = q
10  return r and s
```

0-1 knapsack problem We are given n items. Each item i has a value of v_i dollars and a weight w_i pounds which are nonnegative integers. Given a bag whose capacity is W what is the maximum value that we can carry knowing that we can either carry an item (therefore $1 \cdot w_i$) or leave it behind (therefore $0 \cdot w_i$) but not carry a portion of it?

If i is the highest-numbered item in an optimal solution S for W pounds and items $1\,..\,n$. Then $S\prime = S - \{i\}$ must be an optimal solution for $W - w_i$ pounds and items $1\,..\,i$-1 pounds, and the value of the solution S is v_i plus the value of the subproblem solution $S\prime$. Assuming $c[i, w]$ to be the value of the solution for items $1\,..\,1$ and maximum weight w we can formalize the previous statement as

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1,w] & \text{if } w_i > w \\ \max(v_i + c[i-1, w-w_i], c[i-1,w]) & \text{if } i > 1 \text{ and } w \geq w_i \end{cases}$$

The following dynamic programming algorithm takes as input the arrays of values $v[1\,..\,n]$ and weights $w[1\,..\,n]$, the capacity W and returns a table $c[0\,..\,n, 0\,..\,W]$ which contains the values of the solutions for all subproblems. At the end of the computation $c[n, W]$ contains the maximum value we can carry.

```
0-1-KNAPSACK(v, w, W)
 1  n = v.length
 2  let c[0..n, 0..W] be a new table
 3  for w = 0 to W
 4       c[0, w] = 0
 5  for i = 1 to n
 6       c[i, 0] = 0
 7  for i = 1 to n
 8       for w = 1 to W
 9            if w[i] ≤ w
10                 if v[i] + c[i − 1, w − w[i]] > c[i − 1, w]
11                      c[i, w] = v[i] + c[i − 1, w − w[i]]
12                 else c[i, w] = c[i − 1, w]
13            else c[i, w] = c[i − 1, w]
14  return c
```

To find out which items are included in the solution we need to start at c$[n, W]$ and tracing where the optimal values came from. If $c[i, w] = c[i-1, w]$, then item i is not part of the solution, and we continue tracing with $c[i-1, w]$. Otherwise item i is part of the solution, and we continue tracing with $c[i-1, w-w[i]]$.

6.2 Greedy Algorithms

A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices that looks best at the moment (i.e. a local optimal choice) hoping that it will lead to a globally optimal solution.

An optimization problem needs to have the following two characteristics in order to be efficiently solvable with a greedy algorithm:

- expose *optimal substructure*, exactly as DP
- have a *greedy-choice property*, which means that we can assemble a globally optimal solution by making a locally optimal (greedy) choices.

In DP algorithms we make a choice at each step, but the choice usually depends on the solutions to subproblems. In greedy algorithms, we make whatever choice seems best at the moment and then solve the subproblem that remains.

A DP algorithm proceeds bottom-up, whereas a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, reducing each given problem instance to a smaller one.

6.2.1 Proving correctness and optimality

How can we prove that the solution found by a greedy algorithm is optimal if we don't even know what the optimal solution is? Of course if we can prove that an optimal solution exists and is the same one found by the greedy algorithm then we are done. There are cases where this is not possible and therefore we need to use techniques that only use a property of the optimal solution which can in turn be used to prove the correctness and optimality of a greedy algorithm.

- Greedy algorithm stays ahead. Show that after each step of the algorithm, the solution it finds is at least as good as any other algorithm's solution.
- Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.
- Structural. Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

6.2.2 Huffman codes

Huffman's greedy algorithm builds a variable length binary character code, in which each character is represented by a unique binary string, which we call codeword. The goal is to give more frequent character short codewords and infrequent characters long codewords to achieve data compression.

In the pseudocode that follows , we assume C is a set of n characters and that each character $c \in C$ is an object with an attribute $c.freq$ giving its frequency. The algorithm builds a tree T corresponding to the optimal code in a bottom-up fashion and uses a min-priority queue Q, keyed on the $freq$ attribute, to identify the two least-frequent objects to merge together.

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       z.left = x = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  return EXTRACT-MIN(Q)    // returns the root of the tree
```

Running time of this algorithm is $O(n \log n)$ since priority queue operations run in $\log n$ time and the main loop executes $n - 1$ times.

6.2.3 Other problems that are solvable with greedy algorithms

Activity selection problem In this problem we are given a set of activities $A = \{a_1, \ldots, a_n\}$ and their start and finish times s_i and f_i and we need to find the maximum subset of mutually compatible activities. Two activities i and j are compatible if they don't overlap.

To solve this problem we consider activities in increasing order of finish time and we schedule them provided they are compatible with the ones already taken. Here S is the set of selected activities.

```
ACTIVITY-SELECTION(A)
1  SORT(A)        // in increasing order of finish time f_i
2  S = ∅
3  for i = 1 to n
4       if a_i is compatible with S
5            S = S ∪ a_i
6  return S
```

Activity partitioning problem In this problem we are given a set of activities $A = \{a_1, \ldots, a_n\}$ and their start and finish times s_i and f_i and we need to find the minimum number of "slots" necessary to schedule all activities. Clearly two activities can't overlap in the same slot.

To solve this problem we consider activities in increasing order of start time. Here s is the number of a slots.

```
ACTIVITY-PARTITIONING(A)
1  SORT(A)        // in increasing order of start time s_i
2  s = 0
3  for i = 1 to n
4       if a_i is compatible with some slot k
5            schedule a_i in slot k
6       else allocate a new slot s + 1
7            schedule a_i in slot s + 1
8            s = s + 1
9  return s
```

Lateness minimization problem In this problem we are given a set of activities $A = \{a_1, \ldots, a_n\}$, their start and finish times s_i and f_i and their due time (deadline) d_i. We can also define, for each job i, lateness as $l_i = \max(0, f_i - d_i)$ and the amount of time required to complete as $t_i = f_i - s_i$.

Our goal is to schedule all activities minimizing maximum lateness $L = \max_{i \in \{[1..n]\}} l_i$

To solve this problem we consider activities in increasing order of deadline (earlies deadline activities come first). At the end of the algorithm A contains the scheduled activities.

LATENESS-MINIMIZATION(A)

```
1  SORT(A)        // in increasing order of deadline d_i
2  t = 0
3  for i = 1 to n
4       schedule a_i in interval [t, t + t_i]
5       s_i = t
6       f_i = t + t_i
7       t = t + t_i
```

Optimal off-line caching problem In this problem we are given a cache with capacity k and a sequence of requests $d_1, d_2, \ldots, d_m$. We want to find an eviction schedule (i.e. decide which element in the cache we want to overwrite) that minimizes the number of cache misses.

The optimal solution is to evict the item that is not requested until *farthest in the future.*

Coin changing problem In this problem we are given the currency denominations $D = \{1, 5, 10, 25, 100\}$ (US coinage) and and a certain amount x. We need to devise a method to pay the amount x using the fewest number of coins.

The problem can be solved using the so called cashier's algorithm, which at each iteration adds the coin of largest value that does not take us past the amount to be paid.

COIN-CHANGE(D)

```
1  SORT(D)        // in increasing order
2  S = ∅
3  while x ≠ 0
4       let k be the largest integer such that c_k < x
5       if k == 0
6            return error
7       x = x − c_k
8       S = S ∪ {k}
9  return S
```

Note that the previous algorithm is optimal for US coinage. If we use different denomiations we might need a different algorithm that uses Dynamic Programming.

Fractional knapsack problem In this problem we are given a set of items $A = \{a_1, \ldots, a_n\}$, each with a weight w_i and a value v_i and a container of capacity x. We need to find the set of items (or fraction of items) so that the total weight is less than x and the total value is as large as possible.

The problem can be solved by sorting the items in decreasing order of revenue $r_i = \frac{v_i}{w_i}$ and then filling the container up to x by taking a fraction of the last item if such an item doesn't fit entirely.

7 Graphs[7]

7.1 Data structures

7.1.1 Adjacency-list

The *adjacency-list representation* of a graph $G = (V, E)$ consists of an array Adj of n linked lists, one for each vertex $u \in V$, such that $Adj[u]$ contains all the vertices adjacent to u in G.

- If G is directed then the sum of the lengths of all the adjacency lists is m.
- If G is undirected then the sum of the lengths of all the adjacency lists is $2m$.
- The amount of memory required to store the adjacency-list representation of G is $\Theta(n + m)$: this is the major advantage of this representation.
- The amount of time needed to scan the entire data structure is $\Theta(n + m)$: this is an advantage of this representation.
- The amount of time to search for v in the adjacency-list $Adj[u]$ is $O(\deg(u))$ because in the worst case we need to scan $G.Adj[u]$ entirely.
- The amount of time to check if a given edge (u, v) is present in the graph is $O(\deg(u))$ that is the same time needed to search for v in $Adj[u]$: this is the major drawback of this representation.
- Generally speaking this representation is better for sparse and/or big graphs.

7.1.2 Adjacency-matrix

The *adjacency-matrix representation* of a graph $G = (V, E)$ consists of a matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

- If G is undirected then $A = A^T$, that is A is symmetrical.
- The amount of memory required to store the adjacency-matrix representation is $\Theta(n^2)$, independently from the number of edges: this is the major drawback of this representation
- The amount of time needed to scan the entire data structure is $\Theta(n^2)$ that is asymptotically more complex than the time needed to scan the adjacency-list: this is a drawback of this representation.
- The amount of time to check if a given edge (u, v) is present in the graph is $\Theta(1)$ because we just need to check the relative entry in the adjacency-matrix: this is the major advantage of this representation.
- Generally speaking this representation is better for dense and/or small graphs.

7.2 Elementary Algorithms

7.2.1 Breadth-first search (BFS)

Given a graph $G = (V, E)$ and a distinguished source vertex s this algorithm:

- assumes an adjacency-list representation;
- works both on directed and undirected graphs;
- systematically explores the edges of G to "discover" every vertex that is reachable from s (can be used to identify the connected components if executed on multiple sources);

[7] Chapters 22, 23, 24, 25

- computes the shortest-path[8] from s to every reachable vertex v;
- produces a *breadth-first tree* with root s that contains all reachable vertices (which can be used to print the (reverse-)shortest-path from s to any vertex v);
- uses a *queue* to store the neighbors (which ensures that the algorithm discovers all the vertices at distance k from s before discovering vertices at distance $k+1$;
- has a running time of $O(n+m)$, that is a time linear in the size of the adjacency-list representation (because each vertex is enqueued/dequeued only once and the adjacency-list of each vertex is scanned only when the vertex is dequeued.

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}     // initialize vertices
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY                    // inintializes Q with s
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)               // fetch next vertex to be visited
12       for each v ∈ G.Adj[u]        // add neighbors to queue
13            if u.color == WHITE
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)
18       u.color = BLACK              // mark as visited
```

7.2.2 Depth-first search (DFS)

Given a graph $G = (V, E)$ this algorithm:

- assumes an adjacency-list representation;
- works both on directed and undirected graphs;
- systematically explores the edges out of the most recently discovered vertex v that still has unexplored edges leaving it and, once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered;
- produces a *depth-first forest* $G_\pi$comprising several *depth-first trees*;
- it's recursive, which means it implicitly uses a *stack* to store the vertices that have been *discovered* but not yet *finished*;
- has a running time of $O(n+m)$, that is a time linear in the size of the adjacency-list representation (because the "visit procedure" is called once for every vertex and has a running time of $\sum_{v \in V} |Adj[v]| = \Theta(m)$.

[8] We define the *shortest-path distance* $\delta(s, v)$ from s to v as the minimum number of edges in any path from vertex s to vertex v; if there is no path from s to v then $\delta(s, v) = \infty$.
We call a simple path of length $\delta(s, v)$ from s to v a *shortest path* from s to v.
Note that this is a particular case of the more general shortest-path problem which involves edges with different weights.

```
DFS(G)
1  for each vertex u ∈ G.V      // initialize vertices
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each u ∈ G.V             // make sure we visit all nodes
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
 1  time = time + 1
 2  u.d = time                  // set discovery time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]       // add neighbors to stack
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK             // mark as visited
 9  time = time + 1
10  u.f = time                  // set finishing time
```

Edge classification DFS can be used to classify the edges of the input graph G. We can define four edges types in terms of the depth-first forest G_π:

- *Tree edges* are edges in G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v), which means $v.color = \text{WHITE}$.
- *Back edges* are those edges (u, v) connecting a vertex u to an ancestor v in G_π. Edge (u, v) is a back edge if $v.color = \text{GRAY}$ when v was first discovered by exploring edge (u, v).
- *Forward edges* are those nontree edges (u, v) connecting a vertex u to a descendant v in G_π. Edge (u, v) is a forward edge if $v.color = \text{BLACK}$ and $u.d < v.d$ when v was first discovered by exploring edge (u, v).
- *Cross edges* are all other edges. They can go between vertices in different dept-first trees or vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other. Edge (u, v) is a cross edge if $v.color = \text{BLACK}$ and $u.d > v.d$ when v was first discovered by exploring edge (u, v).

In undirected graphs, since (u, v) and (v, u) are the same edge we classify the edges as the first type in the classification list that applies. It can be shown that in a DFS of an undirected graph G, every edge of G is either a tree edge or a back edge, which means that forward and cross edges never occur.

A directed graph is acyclic if and only if a DFS yields no back edges.

Parenthesis structure In any DFS of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

- the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in G_π.
- the interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$ and u is a descendant of v in G_π.
- the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$ and v is a descendant of u in G_π.

7.2.3 Tree traversal

Since trees are a particular kind of graph we can use both BFS and DFS on them. There are however some notation consideration that need to be done.

- BFS is also called *level-order traversal* since, once we apply such an algorithm starting at the root of a tree, we visit every node on a level before going to a lower level. (since the neighbor of any node are simply its children and its parent which we have already visited)
- Once we implement DFS-Visit(G, n) we have three significant choices when picking the order in which we can visit the current node and its neighbors. This choice has practical consequences when we run DFS on a tree starting from the root.
  - *Preorder visit*: we access the node before visiting all of its neighbor
  - *Inorder visit*: we access the node after visiting some of its neighbor and before visiting some others (several policies can be used)
  - *Postorder visit*: we access the node *after* visiting all of its neighbor, while recursion is returning
- Traversing a tree always costs $\Theta(n)$.

7.2.4 Topological Sort

A *topological sort* of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v) then u appears before v in the ordering.

Topological-Sort(G)
1 call DFS(G) to compute finishing times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

Time complexity We can perform a topological sort in time $\Theta(n + m)$, since DFS takes $\Theta(n + m)$ time and it takes $O(1)$ time to insert each of the n vertices onto the front of the linked list.

7.2.5 Strongly connected components

The key idea behind this algorithm is that G and G^T have the same connected components, which means that u and v are reachable from each other in G if and only if they are reachable from each other in G^T

Strongly-Connected-Components(G)
1 call DFS(G) to compute finishing times $v.f$ for each vertex v
2 compute G^T
3 call DFS(G^T), but in the main loop of DFS, consider the vertices in order
  of decreasing $v.f$ (as computed in line 1)
4 **return** the vertices of each tree in the depth-first forest formed in line 3 as
  a separated strongly connected component

Time complexity We can find the strongly connected components of a directed graph in $O(n+m)$, since the time needed to create G^T is $O(n + m)$ and DFS runs in $\Theta(n + m)$.

7.3 Minimum Spanning Trees

The following algorithms are a classic example of greedy algorithms.

The problem they try to solve is : given a connected, undirected graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$, we wish to find a minimum spanning tree for G.

7.3.1 Kruskal's algorithm

Kruskal's algorithm:

- finds a safe edge to add to the growing forest by finding, of all edges that connect any two trees in the forest, an edge (u, v) of least weight.
- qualifies as a greedy algorithm because at each step it adds to the forest an edge of least possible weight.

Implementation A classic implementation of Kruskal's algorithm uses the union-find data structure to maintain several disjoint sets of elements.

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V      // initialize the data structure
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```

Time complexity (analysis) The running time of Kruskal's algorithm clearly depends on the union-find data structure operations. We perform a total of n MAKE-SET operations (lines 2-3) and $O(m)$ UNION and FIND-SET operations (lines 5-8). The total time required to perform such operations is $O((n+m)\alpha(n))$ (see theorem on cost of union-find operations).

Recall that we assumed that G is connected, which means that $m \geq n - 1$ and that $\alpha(n) = O(\log n) = O(\log m)$. We can then write $O((n + m)\alpha(n)) = O(m\alpha(n)) = O(m \log m)$.

Now, recall that $m < n^2$ for any graph, which implies $\log m = O(\log n^2) = O(2 \log n) = O(\log n)$. We can then write $O(m \log m) = O(m \log n)$.

Time complexity (conclusion) Provided that the edges are *already sorted* or can be sorted in linear time (for instance with radix sort), then the time complexity of Kruskal's algorithm is $O(m\alpha(n))$, $O(m \log n)$ otherwise.

7.3.2 Prim's algorithm

Prim's algorithm:

- has the property that the edges in the set A always form a single tree. The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V.
- qualifies as greedy algorithm because at each step it adds to the tree A an edge that contributes the minimum amount possible to the tree's weight and connects A to an isolated vertex.

Implementation A classic implementation of Prim's algorithm stores all the vertices v that are not in the tree in a min-priority queue Q based on a *key* attribute which represents the minimum weight of any edge connecting v to a vertex in the tree; by convention $v = \infty$ if there is no such edge.

The algorithm implicitly maintains the set A as $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$; therefore, when the algorithm terminates the MST A for G is $A = \{(v, v.\pi) : v \in V - \{r\}\}$.

```
MST-PRIM(G, w, r)
 1  for each vertex u ∈ G.V        // initialize the vertices
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0                      // initializes the root
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each vertex v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

Time complexity The total time required for all calls to EXTRACT-MIN is $O(n \log n)$, since the main loop executes n times and EXTRACT-MIN operation takes $O(\log n)$ time.

The total time required by the inner loop is $O(m \log n)$, since the loop executes $2m$ times (length of the adjacency list) and the time required to restructure the heap (that needs to be done every time a key is modified) is $O(\log n)$.

Therefore the total time for Prim's algorithm is $O((n + m) \log n) = O(m \log n)$. (Recall that the graph is connected!)

7.4 Single-Source Shortest Paths

In a *shortest-path problem* we are given a weighted, directed graph $G = (V, E)$, with weight function $w : E \to \mathbb{R}$ mapping edges to real-valued weights. The weight $w(p)$ of path $p = \langle v_1, v_2, \ldots, v_k \rangle$ is the sum of the weights of its constituent edges: $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$. We define the *shortest-path weight* $\delta(u, v)$ from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path form } u\text{to } v \\ \infty & \text{otherwise} \end{cases}$$

A s*hortest path* from vertex u to vertex v is then defined as any path p with weight $\delta(u, v)$. There are several variants of the shortest-path problem including:

Single-source shortest path: given a graph we want to find the shortest path from a given *source* vertex s to all other vertices.

Single-destination shortest path: given a graph we want to find the shortest path from a given *destination* vertex t to all other vertices. We solve this problem solving single-source shortest path on G^T.

Single-pair shortest path: given a graph we want to find the shortest path from u to v for given vertices u and v. We solve this problem by solving single-source shortest path with source u. Moreover, all known algorithms for this problem have the same worst-case asymptotic running time as the best single-source algorithms.

All-pairs shortest path: given a graph we want to find the shortest path from u to v for every pair of vertices u and v. We solve this problem either solving single-source shortest path with source u for every choice of u or we Floyd-Warshall algorithm (or equivalent).

7.4.1 Cycles and negative-weight edges

- If the graph $G = (V, E)$ contains no *negative-weight cycles* reachable from the source s, then for all $v \in V$, the shortest path weight $\delta(s, v)$ remains well defined, even if it has a negative value.
- A shortest path cannot contain a positive-weight cycle, since removing the cycle from the path produces a path with the same source and destination vertices and a lower path weight.

- Since any acyclic path in a graph G contains at most $|V|$ distinct vertices, it also contains at most $|V| - 1$ edges. Thus, we can restrict out attention to shortest paths of at most $|V| - 1$ edges.

7.4.2 Path representation and relaxation

We represent shortest paths similarly to how we represented breadth-first trees; we maintain, for every vertex v, a predecessor $v.\pi$ that is either another vertex or NIL. This will produce, once the algorithms terminate, a predecessor subgraph G_π which represents the shortest-path tree.

We maintain an attribute $v.d$ for every vertex v which is an upper bound on the weight of a shortest path from source s to v. We call $v.d$ a shortest path estimate.

We initialize $v.\pi$ and $v.d$ with the following $\Theta(n)$ procedure.

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating $v.d$ and $v.\pi$. The following code performs the relaxation step on edge (u, v) in $O(1)$ time.

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

7.4.3 The Bellman-Ford algorithm

The Bellman-Ford algorithm solves the single-source shortest path problem in the general case in which edge weights may be negative.

Given a weighted, directed graph G with source s and weight function, the algorithm returns a Boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Time complexity The Bellman-Ford algorithm runs in time $O(nm)$, since the initialization in line 1 takes $\Theta(n)$ time, each of the $|V| - 1$ passes over the edges in lines 2-4 takes $\Theta(m)$ time, and the for loop of lines 5-7 takes $O(m)$ time.

7.4.4 Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest path problem on a weighted graph G for the case in which all edge weights are nonnegative. Therefore, we assume that $w(u, v) \geq 0$ for each edge.

Dijkstra's algorithm maintains a set S of vertices whose final shortest path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum

shortest path estimate, adds u to S and relaxes all edges leaving u. Because Dijkstra's algorithm always chooses the "lightest" (or "closest") vertex in $V - S$ to add to set S, we say that it uses a greedy strategy.

The following implementation uses a min-priority queue Q of vertices keyed by their d values.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Time complexity Because each vertex u is added to the set S exactly once the while loop in lines 4-8 is executed exactly $|V|$ times. Also, each edge is examined exactly once in the for of lines 7-8, which means that this loop iterates a total of $|E|$ times.

If we implement the priority queue using an array, each INSERT and RELAX operation takes $O(1)$ time, and each EXTRACT-MIN operations takes $O(n)$ time (because we have to scan the whole array). Thus, the running time of Dijkstra's algorithm is $O(n^2 + m) = O(n^2)$ (since $m \leq n^2$ in any graph).

Vice-versa, if we implement the priority queue using a min-heap we know that both EXTRACT-MIN and RELAX need $O(\log n)$. Thus, the total running time of Dijkstra's algorithm is $O((n + m) \log n)$, which is $O(m \log n)$ if all the vertices are reachable from the source. This running time improves upon the straightforward array implementation only if the graph is sufficiently sparse (i.e. $m = o(\frac{n^2}{\log n})$.

7.5 All-Pairs shortest Paths

7.5.1 Iterated Dijkstra and Bellman-Ford's algorithms

We can solve an all-pair shortest path problem by running a single-source shortest paths algorithm n times.

If all edged are nonnegative, we can use Dijkstra's algorithm. If we use the linked list implementation, the running time is $O(n^3 + nm)$. If we use the min-heap implementation, instead, the running time is $O(nm \log n)$.

If we have negative-weight edges we need to iterate Bellman-Ford algorithms which yields a time complexity of $O(n^2 m)$ which on a dense graph is $O(n^4)$.

7.5.2 Floyd-Warshall's algorithm

Unlike the majority of graph algorithms, Floyd-Warshall algorithm assumes that the input graph $G = (V, E)$ is represented using an adjacency matrix representation. For convenience, we assume that the vertices are numbered $1, 2, \ldots, n$, so that the input is an $n \times n$ matrix $W = (w_{ij})$ representing the edge weights and defined as follows

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E \end{cases}$$

We *allow negative-weight edges*, but we assume that the input graph contains *no negative-length cycles.*

The output of of the algorithm is the $n \times n$ matrix $D = (d_{ij})$, where entry d_{ij} contains the weight of a shortest path from vertex i to vertex j. That is, if we let $\delta(i, j)$ denote the shortest path weight from vertex i to vertex j, then $d_{ij} = \delta(i, j)$ at termination.

To solve the all-pair shortest path problem on an input adjacency matrix, we need to compute not only the shortest path weights but also the predecessor matrix $\Pi = (\pi_{ij})$, where π_{ij} is NIL if either $i = j$ or there is no path from i to j, and otherwise π_{ij} is the predecessor of j on some shortest path from i.

Implementation The Floyd-Warshall algorithm takes a completely different approach and relies on the following observation.

For any pair of vertices $i, j \in V$, consider all shortest paths from i to j whose intermediate vertices are drawn from a subset $\{1, 2, \ldots, k\}$, and let p be a minimum-weight path from among them.

If k is not an intermediate vertex on path p, then all intermediate vertices of path p are in the set $\{1, 2, \ldots, k-1\}$. Thus a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$ is also a shortest path from i to j with all intermediate vertices in $\{1, 2, \ldots, k\}$.

If k is an intermediate vertex on path p, then we decompose p into $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$. p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k\}$. Because vertex k is not an intermediate vertex of path p_1, all intermediate vertices of p_1 are in the set $\{1, 2, \ldots, k-1\}$. Since subpaths of shortest paths are shortest paths, p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Similarly p_2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$.

Let $d_{ij}^{(k)}$be the weights of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$. When $k = 0$, a path from i to j has no intermediate vertices at all (i.e. it has at most one edge) and therefore $d_{ij}^{(0)} = w_{ij}$. Based on the previous observations we can define $d_{ij}^{(k)}$ recursively by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min(d_{ij}^{(k)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1 \end{cases}$$

Because for any path, all intermediate vertices are in the set $\{1, 2, \ldots, n\}$, the matrix $D^{(n)} = (d_{ij}^{(n)})$ gives the final answer: $\delta(i, j)$ for all $i, j \in V$.

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4       let D^(k) = (d_ij^(k)) be a new n × n matrix
5       for j = 1 to n
6            for j = 1 to n
7                 d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

Time complexity Floyd-Warshall algorithm runs in $\Theta(n^3)$ because of the triple nested loop.

7.6 Longest path problem and other time complexity considerations

- We have seen that the shortest path problem, in all of its variants, can be solved in polynomial time (it is in P) in graphs without negative-weight cycles.
- If the graph *has* negative-weight cycles, the shortest path problem becomes NP-complete. This means we can still run Bellman-Ford algorithm on such a graph but we cannot guarantee it will produce a correct answer if a negative-weight cycle is reachable from the source, even if the algorithm can detect them.
- The *longest path problem* can be reduced to the shortest path problem, by exploiting the duality of optimizations (maximizing a positive value is the same as minimizing a negative value). If the input graph to the longest path problem is G, the shortest path on the graph H, which is exactly the same as G but with edge weights negated, is the longest path on G. However, any positive-weight cycle in the original graph G leads to a negative-weight cycle in H and, therefore, solving the shortest path problem for G is also NP-complete.

- If G contains no cycles, then H will have no negative-weight cycles, and we can use any of the algorithms we have seen to solve the shortest-path problem. Thus the longest path problem is easy on acyclic graphs(including dags).

7.7 Oher graph algorithms

7.7.1 Testing bipartitness

We can use BFS to do so. If at the end of BFS there are no intra-layer edges then the graph is bipartite.

A Lists and tables

A.1 Growth rates of different functions

The following table summarizes the asymptotical growth rates of the most common classes of functions in algorithm analysis. They are listed in order from the *slower* to the *quicker* asymptotically growing.

Name	Growth rate	Function class	Example
Constant	$\Theta(1)$	c	10
Inverse Ackermann	$\Theta(\alpha(n))$	-	-
Iterated logarithm	$\Theta(\log^* n)$	$\log^* n$	-
Log-logarithmic	$\Theta(\log\log n)$	$\log\log n$	-
Logarithmic	$\Theta(\log n)$	$\log(poly(n))$	$\log n$, $\log n^2$
Polylogaritmic	$\Theta(\log^c n)$ with $c > 1$	$poly(\log n)$	$\log^2 n$
Fractional power (or Sublinear)	$\Theta(n^c)$ with $0 < c < 1$	n^c	$n^{\frac{2}{3}}$
— Linear —	$\Theta(n)$	n	-
Linearitmic	$\Theta(n\log n)$	-	$n\log n$
Polynomial	$\Theta(n^c)$ with $c > 1$	$poly(n)$	n^2, n^3
Exponential	$\Theta(c^n)$ with $c > 1$	$c^{poly(n)}$	10^n, e^n
Factorial	$\Theta(n!)$	$n!$	$n!$
Double Exponential	$\Theta(c^{2^{poly(n)}})$ with $c > 1$	$c^{2^{poly(n)}}$	2^{3^n}

Note that for any polylogaritmic function we can ignore the lower order terms when deriving the asymptotic growth rate.

Also note that sometimes we say that linear and linearitmic functions have a growth rate that is *polynomial*: use your brain to extract the proper meaning of the word polynomial from the context!

A.2 Complexities for basic data structures operations

N/A means that the operation in not applicable to the data structure.

D means that it depends, look at bottom of each table for explanation. In addition to these tables check out http://bigocheatsheet.com/ for an excellent summary of complexities of basic data structures and algorithms.

A.2.1 Linear data structures

Data structure	Implementation	Access	Insert	Delete	Peek
List (index known)	array	$O(1)$	$O(n)$	$O(n)$	N/A
	linked list	$O(1)$	$O(1)$	$O(1)$	N/A
Stack	array	N/A	$O(1)$	$O(1)$	$O(1)$
	linked lists	N/A	$O(1)$	$O(1)$	$O(1)$
Queue	array	N/A	$O(1)$	$O(1)$	$O(1)$
	linked lists	N/A	$O(1)$	$O(1)$	$O(1)$

Note how the costs for array and linked list operations refer to the case when the "position" is known, that is the index for the arrays and the pointer to an element for linked lists.

A.2.2 Trees

Data structure	Impl.	Insert	Delete	Search	Build	Max/Min	Pred./Suc.
Heap	array	$O(\log n)$	$O(\log n)$	N/A	$O(n)$	D	N/A
Priority q.	heap	$O(\log n)$	$O(\log n)$	N/A	$O(n)$	D	N/A
BST	3-pointers	$O(h)$	$O(h)$	$O(h)$	$O(nh)$	$O(h)$	$O(h)$
Bal. BST	3-pointers	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n\log n)$	$O(\log n)$	$O(\log n)$

Note that min-heap (max-heap) support only min-priority queues (max-priority queues); therefore only $\min(Q)$ ($\max(Q)$) is supported in $O(1)$ time.

A.2.3 Disjoint-set data structure

Data structure	Implementation	Make-Set	Union	Find-Set
Union-Find	Linked lists	$O(1)$	$\Omega(n)$	$O(1)$
Union-Find	Trees	$O(1)$	$O(\alpha(n))$	$O(\alpha(n))$

A.3 Sort, selection and search algorithms complexities

Complexities of the main sorting and selection algorithms.

Algorithm	Worst Case	Best Case	Avg. case	In place	Notes
Insertion sort	$O(n^2)$	$O(n)$	$O(n^2)$	Yes	-
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No	-
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes	-
Quick sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	Yes	-
Quick select	$O(n^2)$	$O(n)$	$O(n)$	N/A	-
Binary search	$O(\log n)$	$O(1)$	$O(\log n)$	N/A	using balanced BSTs

A.4 Graph algorithms complexities

Remember that, in any graph, $m = O(n^2)$[9].

Algorithm	Data struct.	Running time	Notes
BFS	-	$O(n + m)$	-
DFS	-	$O(n + m)$	-
Topological Sort	-	$O(n + m)$	-
Prim	min-heap	$O((n + m) \log n)$	$O(m + n \log n)$ with Fibonacci heap
Kruskal	union-find	$O(m \log n)$	-
Bellman-Ford	min-heap	$O(nm)$	-
Dijkstra	array	$O(n^2 + m) = O(n^2)$	-
Dijkstra	min-heap	$O((n + m) \log n)$	$O(m + n \log n)$ with Fibonacci heap
Iterated Dijkstra	array	$O(n^3 + nm) = O(n^3)$	-
Iterated Dijkstra	min-heap	$O(nm \log n)$	-
Iterated B-F	min-heap	$O(n^2 m)$	-
Floyd-Warshall	array	$O(n^3)$	-

Note also how in connected graphs we have $m \geq n - 1$ and therefore $O((n + m) \log n) = O(m \log n)$.

[9] Therefore $O(\log m) = O(\log n^2) = O(2 \log n) = O(\log n)$