4 Software implementation

This chapter covers

- Data structures: linear, nonlinear, and probabilistic
- Problem-solving paradigms: complete search, greedy, divide and conquer, and dynamic programming
- ML research: sampling methods and variational inference

In the previous chapters, we looked at two main camps of Bayesian inference: Markov chain Monte Carlo and variational inference. In this chapter, we review computer science concepts required for implementing algorithms from scratch. To write high-quality code, it's important to have a good grasp of data structures and algorithm fundamentals. This chapter is designed to introduce common computational structures and problem-solving paradigms. Many of the concepts reviewed in this section are interactively visualized on the VisuAlgo website (https://visualgo.net/en).

4.1 Data structures

A *data structure* is a way of storing and organizing data. Data structures support several operations, such as insertion, search, deletion, and updates, and the right

choice of data structure can simplify the runtime of an algorithm. Each data structure offers different performance tradeoffs. As a result, it's important to understand how data structures work.

4.1.1 Linear

A data structure is considered *linear* if its elements are arranged in a linear fashion. The simplest example of a linear data structure is a *fixed-size array* (where the size of the array may be specified as a constraint of the problem). The time it takes to access an element in an array is constant: $O(1)$. If the size of the array is not known ahead of time, it's best to use a *dynamically resizable array* (e.g., a List in Python or a Vector in C++), as these data structures are designed to handle resizing natively.

Two common operations applied to arrays are searching and sorting. The simplest search is a linear scan through all elements in $O(n)$ time. If the array is sorted, we can use binary search in $O(\log n)$ time, which is an example of a divide-and-conquer algorithm, which we will discuss soon. Naive array sorting algorithms, such as selection sort and insertion sort, have a complexity of $O(n^2)$ and are, therefore, only suitable for small inputs. In general, comparison-based sorts where elements are compared pairwise, such as merge, heap, or quicksort, have the runtime of $O(n \log n)$ because the time complexity can be thought of as traversing a complete binary tree, where each leaf represents one sorted ordering. In this representation, the height of the tree h is equal to the algorithm runtime. Since there are $n!$ possible orderings (leaf nodes), we can bound the runtime as follows.

$$\begin{aligned} h &= \log n! = \log n(n-1)(n-2) \times \cdots \times 1 \\ &= \log n + \log(n-1) + \cdots + \log 1 \\ &< \log n + \log n + \cdots + \log n = n \log n = O(n \log n) \end{aligned} \tag{4.1}$$

If we add additional constraints on the input, we can construct linear-time $O(n)$ sorting algorithms, such as count sort, radix sort, and bucket sort. For example, count sort can be applied to integers in a small range, and radix sort works by applying count sort digit by digit, as discussed in *Competitive Programming* by Steven Halim (2010). Making algorithms distributed can further reduce runtime as described in *The Art of Multiprocessor Programming* by Mauice Herlihy and Nir Shavit (2012).

A *linked list* consists of nodes that store a value and a next pointer, starting from the head node. It's usually avoided due to its linear $O(n)$ search time. A *stack* allows $O(1)$ insertions (push) and $O(1)$ deletions (pop) in the last-in, first-out (LIFO) order. This is particularly useful in algorithms that are implemented recursively (e.g., bracket matching or topological sort). A *queue* allows $O(1)$ insertion (enqueue) from the back and $O(1)$ deletion (dequeue) from the front; thus, it follows the first-in, first-out (FIFO) model. It's a commonly used data structure in algorithms such as breadth-first search (BFS) and those based on BFS. In the next section, we will define and examine several examples of nonlinear data structures.

4.1.2 Nonlinear

A data structure is considered *nonlinear* if its elements do not follow a linear order. Examples of nonlinear data structures include a *map* (dictionary) and a *set*. This is because ordered dictionaries and ordered sets are built on self-balanced binary search trees (BST) that guarantee $O(n \log n)$ insertion, search, and deletion operations. BSTs have the property that the root node value is greater than its left child and less than its right child for every subtree. Self-balanced BSTs are typically implemented as Adelson-Velskii-Landis (AVL) or red–black (RB) trees (see *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein [1990] for details). The difference between an ordered map (dictionary) and ordered set data structures is that the map stores (key–value) pairs, while the set only stores keys. A *heap* is another way to organize data in a tree representation. For example, an array A = [2, 7, 26, 25, 19, 17, 1, 90, 3, 36] can be represented as a tree, as shown in figure 4.1.

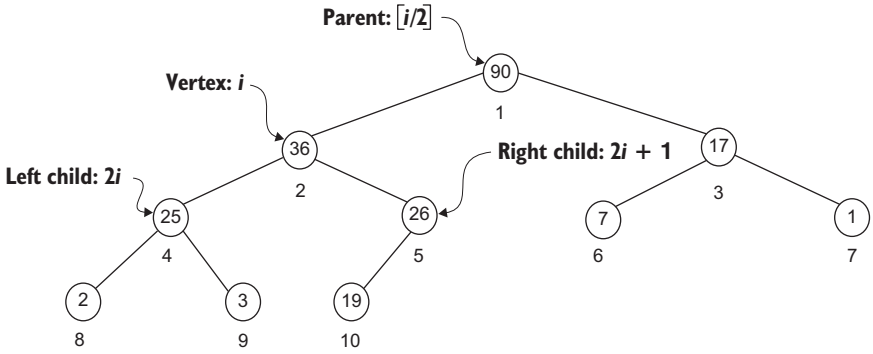

Figure 4.1 A binary heap

We can easily navigate the binary heap A = [90, 36, 17, 25, 26, 7, 1, 2, 3, 19], starting with a vertex i and using the simple index arithmetic $2i$ to access the left child, $2i+1$ to access the right child, and $i/2$ to access the parent node. Instead of enforcing the binary search tree (BST) property, the (max) heap enforces the heap property: in each subtree rooted at *x*, items on the left and right subtrees of *x* are smaller than (or equal to) *x*. This property guarantees that the top of the (max) heap is always the maximum element. Thus, (max) heap allows for fast extraction of the maximum element. Indeed, extract max and insert operations are achieved in $O(\log n)$ tree traversal, performing swapping operations to maintain the heap property whenever necessary. A heap forms the basis for a *priority queue*, which is an important data structure in algorithms such as Prim and Kruskal minimum spanning trees (MST), Dijkstra's single-source shortest paths (SSSP), and the A* search. Finally, a *hash table* or unordered map is a very efficient data structure, with $O(1)$ access assuming no collisions. One commonly used class of hash tables is direct addressing (DA), where the keys themselves are the indices. The goal of a hash function is to uniformly distribute the elements in

the table so as to minimize collisions. On the other hand, if you are looking to group similar items in the same bucket, locality-sensitive hashing (LSH) allows you to find nearest neighbors, as described in *Beyond Locality-Sensitive Hashing* by Alexandr Andoni Piotr Indyk, Huy Le Nguyen, and Ilya Razenshteyn (SODA, 2014).

4.1.3 Probabilistic

Probabilistic data structures are designed to handle big data. They provide probabilistic guarantees and result in drastic memory savings. Probabilistic data structures tackle the following common big data challenges:

- Membership querying: bloom filter, counting bloom filter, quotient filter, and cuckoo filter
- Cardinality: linear counting, probabilistic counting, LogLog, HyperLogLog, and HyperLogLog++
- Frequency: majority algorithm, frequent, count sketch, and count-min sketch
- Rank: random sampling, q-digest, and t-digest
- Similarity: LSH, MinHash, and SimHash

For a comprehensive discussion on probabilistic data structures, please refer to *Probabilistic Data Structures and Algorithms for Big Data Applications* by Andrii Gakhov (2022). In the next section, we will discuss four main algorithmic paradigms.

4.2 Problem-solving paradigms

There are four main algorithmic paradigms: complete search, greedy, divide and conquer, and dynamic programming. Depending on the problem at hand, the solution can often be found by recalling the algorithmic paradigms. In this section, we'll discuss each strategy and provide an example.

4.2.1 Complete search

Complete search (aka brute force) is a method for solving a problem by traversing the entire search space to find a solution. During the search, we can prune parts of the search space that we are sure do not lead to the required solution. For example, consider the problem of generating subsets. We can either use a recursive solution or an iterative one. In both cases, we terminate when we reach the required subset size. In the following listing, we will implement a complete search strategy based on an example of generating subsets.

Listing 4.1 Subset generation

```
def search(k, n):
    if (k == n):
        print(subset)    <------ process subset
    else:
        search(k+1, n)
        subset.append(k)
```

```
            search(k+1, n)
            subset.pop()
        #end if

def bitseq(n):
    for b in range(1 << n):
        subset = []
        for i in range(n):
            if (b & 1 << i):
                subset.append(i)
        #end for
        print(subset)
    #end for

if __name__ == "__main__":
    n = 4
    subset = []
    search(0, n)    <------ recursive

    subset = []
    bitseq(n)    <------ iterative
```

A machine learning example where a complete search takes place is in exact inference by complete enumeration—for details, see chapter 21 of *Information Theory, Inference and Learning Algorithms* by David J. C. MacKay. Given a graphical model, we would like to factor a joint distribution according to conditional independence relations and use the Bayes rule to compute the posterior probability of certain events. In this case, we need to completely fill out the necessary probability tables to carry out our calculation.

4.2.2 Greedy

A greedy algorithm takes a locally optimum choice at each step, with the hope of eventually reaching a globally optimum solution. Greedy algorithms often rely on a greedy heuristic, and one can often find examples in which greedy algorithms fail to achieve the global optimum. For example, consider the problem of a fractional knapsack. The purpose of a greedy knapsack problem is to select items to place in a knapsack of limited capacity W so as to maximize the total value of knapsack items, where each item has an associated weight and value. We can define a greedy heuristic to be a ratio of item value to item weight (i.e., we would like to greedily choose items that are simultaneously of high value and low weight and sort the items based on this criteria). In the fractional knapsack problem, we are allowed to take fractions of an item (as opposed to 0–1 knapsack). In the following listing, we will implement a greedy strategy based on the fractional knapsack example.

Listing 4.2 Fractional knapsack

```
class Item:
    def __init__(self, wt, val, ind):
        self.wt = wt
```

```
        self.val = val
        self.ind = ind
        self.cost = val // wt

    def __lt__(self, other):
        return self.cost < other.cost

class FractionalKnapSack:
    def get_max_value(self, wt, val, capacity):

        item_list = []
        for i in range(len(wt)):
            item_list.append(Item(wt[i], val[i], i))

        # sorting items by cost heuristic
        item_list.sort(reverse = True)  #O(nlogn)

        total_value = 0
        for i in item_list:
            cur_wt = int(i.wt)
            cur_val = int(i.val)
            if capacity - cur_wt >= 0:
                capacity -= cur_wt
                total_value += cur_val
            else:
                fraction = capacity / cur_wt
                total_value += cur_val * fraction
                capacity = int(capacity - (cur_wt * fraction))
                break
        return total_value

if __name__ == "__main__":
    wt = [10, 20, 30]
    val = [60, 100, 120]
    capacity = 50

    fk = FractionalKnapSack()
    max_value = fk.get_max_value(wt, val, capacity)
    print("greedy fractional knapsack")
    print("maximum value: ", max_value)
```

Since sorting is the most expensive operation, the algorithm runs in $O(n \log n)$ time. We can see that the input items are sorted by decreasing ratio of value/cost; after greedily selecting items 1 and 2, we take a 2/3 fraction of item 3 for a total value of $60 + 100 + (2/3)120 = 240$.

A machine learning example of a greedy algorithm consists of sensor placement. Given a room and several temperature sensors, we would like to place the sensors in a way that maximizes room coverage. A simple greedy approach is to start with an empty set S0 and at iteration `i` add the sensor `A` that maximizes an increment function, such as mutual information $F_{\text{MI}}(A) = H(V \setminus A) - H(V \setminus A|A)$, where V is the set of all sensors. Here, we used the identity $I(X; Y) = H(X) - H(X|Y)$. It turns out that mutual information is *submodular* if observed variables are independent given the latent state, which leads to efficient greedy

submodular optimization algorithms with performance guarantees (e.g., see the *Optimizing Sensing: Theory and Applications* PhD thesis by Andreas Kraus [2008]).

4.2.3 Divide and conquer

Divide and conquer is a technique that divides a problem into smaller, *independent* subproblems and then combines solutions to each of the subproblems. Examples of the divide and conquer technique include sorting algorithms, such as quick sort, merge sort, and heap sort, as well as binary search. A classic use of binary search is searching for a value in a sorted array. In this use case, we first check the middle of the array to see if it contains what we are looking for. If it does or there are no more items to consider, we stop. Otherwise, we decide whether the answer is to the left or right of the middle element and continue searching. As the size of the search space is halved after each check, the complexity of the algorithm is $O(\log n)$. In the following listing, we will implement a divide and conquer strategy based on a binary search example.

Listing 4.3 Binary search

```
def binary_search(arr, l, r, x):
    #assumes a sorted array
    if l <= r:
        mid = int(l + (r-l) / 2)

        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            return binary_search(arr, l, mid-1, x)
        else:
            return binary_search(arr, mid+1, r, x)
        #end if
    else:
        return -1

if __name__ == "__main__":

    x = 5
    arr = sorted([1, 7, 8, 3, 2, 5])

    print(arr)
    print("binary search:")
    result = binary_search(arr, 0, len(arr)-1, x)

    if result != -1:
        print("element {} is found at index {}.".format(x, result))
    else:
        print("element is not found.")
```

A machine learning example that uses the divide and conquer paradigm can be found in the CART decision tree algorithm, in which the threshold partitioning is done in a divide-and-conquer manner, and the nodes are split recursively until the maximum

depth of the tree is reached. In CART algorithm, as we will see in the next chapter, an optimum threshold is found greedily by optimizing a classification objective (e.g., Gini index), and the same procedure is applied on a tree of one depth greater, resulting in a recursive algorithm.

4.2.4 Dynamic programming

Dynamic programming (DP) is a technique that divides a problem into smaller, *overlapping* subproblems, computes a solution for each subproblem, and stores it in a DP table. The final solution is read off the DP table. Key skills in mastering dynamic programming include the ability to determine the problem states (entries of the DP table) and the relationships or transitions between the states. Then, having defined base cases and recursive relationships, one can populate the DP table in a top-down or bottom-up fashion. In top-down DP, the table is populated recursively, as needed, starting from the top and going down to smaller subproblems. In bottom-up DP, the table is populated iteratively, starting from the smallest subproblems and using their solutions to build on and arrive at solutions to larger subproblems. In both cases, if we already encountered a subproblem, we can simply look up the solution in the table (as opposed to recomputing it from scratch). This dramatically reduces the computational cost.

We use binomial coefficients example to illustrate the use of top-down and bottom-up DP. The following code is based on the recursion for binomial coefficients with overlapping subproblems. Let $C(n, k)$ denote n choose k, and then, we have the following equation.

$$\begin{aligned} \text{Base case} &: C(n,0) = C(n,n) = 1 \\ \text{Recursion} &: C(n,k) = C(n-1,k-1) + C(n-1,k) \end{aligned} \tag{4.2}$$

Notice that we have multiple overlapping subproblems. For example, for $C(n{=}5, k{=}2)$, the recursion tree is as shown in figure 4.2. We can implement top-down and bottom-up DP as shown in listing 4.4.

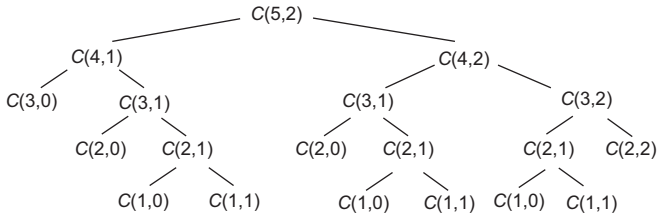

Figure 4.2 Binomial coefficient C(5,2) recursion

Listing 4.4 Binomial coefficients

```
def binomial_coeffs1(n, k):
    #top down DP
    if (k == 0 or k == n):
```

```
        return 1
    if (memo[n][k] != -1):
        return memo[n][k]
    memo[n][k] = binomial_coeffs1(n-1, k-1) + binomial_coeffs1(n-1, k)
    return memo[n][k]

def binomial_coeffs2(n, k):
    #bottom up DP
    for i in range(n+1):
        for j in range(min(i,k)+1):
            if (j == 0 or j == i):
                memo[i][j] = 1
            else:
                memo[i][j] = memo[i-1][j-1] + memo[i-1][j]
            #end if
        #end for
    #end for
    return memo[n][k]
def print_array(memo):
    for i in range(len(memo)):
        print('\t'.join([str(x) for x in memo[i]]))

if __name__ == "__main__":
    n = 5
    k = 2
    print("top down DP")
    memo = [[-1 for i in range(6)] for j in range(6)]
    nCk = binomial_coeffs1(n, k)
    print_array(memo)
    print("C(n={}, k={}) = {}".format(n,k,nCk))

    print("bottom up DP")
    memo = [[-1 for i in range(6)] for j in range(6)]
    nCk = binomial_coeffs2(n, k)
    print_array(memo)
    print("C(n={}, k={}) = {}".format(n,k,nCk))
```

The time complexity is $O(nk)$, and the space complexity is $O(nk)$. In the case of top-down DP, solutions to subproblems are stored (memoized) as needed, whereas in bottom-up DP, the entire table is computed, starting from the base case.

A ML example that uses dynamic programming can be found in reinforcement learning (RL) when finding the solution to Bellman equations. We can write down the value of a state based on its reward at time `t` and the sum of future discounted rewards as shown in the following equation.

$$\begin{aligned} v_\pi(s) &= \Sigma_a \pi(a|s) \Sigma_{r,s'} p(r, s'|s, a)[r + \gamma v_\pi(s')] \\ &= E_\pi[R_t + \gamma v_\pi(S_{t+1})|S_t = s] \end{aligned} \tag{4.3}$$

Equation 4.3 is known as the Bellman optimality equation for $v(s)$. We can recover the optimum policy by solving for an action that maximizes the state reward described by the Q-function.

$$\pi(s) = \arg\max_a q(s, a) \tag{4.4}$$

For a small number of states and actions, we can compute the Q-function in a tabular way, using dynamic programming. In RL, we often want to balance exploration and exploitation—in which case, we take the preceding argmax with a probability of $1 - \epsilon$ and take a random action with probability ϵ.

4.3 ML research: Sampling methods and variational inference

In this section, we focus on ML research, which is a very important skill to have to be current in the field. We focus on the latest developments in the area of sampling methods and variational inference. As we observed in this chapter, many modern ML algorithms include clever algorithms to approximate hard-to-compute posterior densities as a result of intractable, high-dimensional integrals involved in the computation of the posterior.

It is worth spending time to briefly compare the differences between MCMC and variational inference. The advantages of variational inference include that for small to medium problems, it is usually faster, it is deterministic, it is easy to determine when to stop, and it often provides a lower bound on log likelihood. The advantages of sampling include that it is often simpler to implement, it is applicable to a broader range of problems (e.g., problems without nice conjugate priors), and sampling can be faster when applied to very large models or datasets. See chapter 24 of *Machine Learning: A Probabilistic Perspective* by Kevin P. Murphy for additional discussion on the topic.

In addition to the classic MCMC sampling algorithms we studied in previous chapters, a few others deserving attention are slice sampling (see Radford M. Neal's, "Slice Sampling," *Annals of Statistics*, 2003), the Hamiltonian Monte Carlo (HMC; see Radford M. Neal's, "MCMC Using Hamiltonian Dynamics," arXiv, 2012), and the no-U-turn sampler (NUTS; see Matthew D. Hoffman and Andrew Gelman's, "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo," *Journal of Machine Learning Research*, 2014). The state-of-the-art NUTS algorithm is commonly used as an MCMC inference method for probabilistic graphical models and is implemented in the PyMC3, Stan, TensorFlow Probability, and Pyro probabilistic programming libraries.

There have been several attempts to scale MCMC for big data, leading to Stochastic Monte Carlo methods that can be generally grouped into stochastic gradient based methods, methods using approximate Metropolis-Hastings with randomly sampled mini-batches, and data augmentation. Another popular class of MCMC algorithms is streaming Monte Carlo that approximates the posterior for online Bayesian inference. Sequential Monte Carlo (SMC) relies on resampling and propagating samples over time with a large number of particles.

Parallelizing Monte Carlo algorithms is another major area of research. If blocks of independent samples can be drawn from the posterior or a proposal distribution, the sampling algorithm can be parallelized by running multiple independent samplers on separate machines and then aggregating the results. Additional methods include

divide-and-conquer and pre-fetching (see Jun Zhu, Jianfei Chen, and Wenbo Hu, Bo Zhang's, "Big Learning with Bayesian Methods," *National Science Review*, 2017).

Advances in variational inference (VI); span-scalable VI, which includes stochastic approximations; generic VI, which extends the applicability of VI to nonconjugate models; accurate VI, which includes variational models beyond mean-field approximation; and amortized VI, which implements the inference over local latent variables with inference networks (see Cheng Zhang, Judith Butepage, Hedvig Kjellstrom, and Stephan Mandt's, "Advances in Variational Inference," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019). Scalable VI includes stochastic variational inference (SVI), which applies stochastic optimization techniques of the variational objective. The efficiency of stochastic gradient descent (SGD) depends on the variance of gradient estimates (smaller gradient noise allows for larger learning rates and leads to faster convergence). Techniques such as adaptive learning rates and mini-batch size as well as variance reduction, such as control variates, nonuniform sampling, and other approaches, are used to speed up convergence.

In addition to stochastic optimization, leveraging model structure can help achieve the same objective. Examples include collapsed, sparse, parallel, and distributed inference. The collapsed VI relies on the idea of analytically integrating out certain model parameters. Sparse inference exploits either sparsely distributed parameters or datasets that can be summarized by a small number of representative points. In addition, structured VI examines variational distributions that are not fully factorized, leading to more accurate approximations.

Finally, amortized variational inference is an interesting research area that combines probabilistic graphical models and neural networks. The term *amortized* refers to utilizing inference from past computations to support future computations. Amortized inference became a popular tool for inference in deep latent variable models (DLVM), such as the variational autoencoder (VAE; see Diederik P. Kingma and Max Welling's, "Auto-Encoding Variational Bayes," arXiv, 2013). Similarly, neural networks can be used to learn the parameters of conditional distributions in directed probabilistic graphical models (PGM; see Diederik P. Kingma's, "Variational Inference and Deep Learning: A New Synthesis", PhD Thesis, 2017).

4.4 Exercises

4.1 Prove the following binomial identity: $C(n, k) = C(n-1, k-1) + C(n-1, k)$.
4.2 Derive the Gibbs inequality: $H(p, q) \geq H(q)$, where $H(p, q) = -\sum x p(x) \log q(x)$ is the cross-entropy and $H(q) = -\sum q(x) \log q(x)$ is the entropy.
4.3 Use Jensen's inequality with $f(x) = \log(x)$ to prove the AM $\geq$ GM inequality.
4.4 Prove that $I(x; y) = H(x) - H(x|y) = H(y) - H(y|x)$.

Summary

- A data structure is a way of storing and organizing data. Data structures can be categorized into linear, nonlinear, and probabilistic.
- We looked at four algorithmic paradigms in this chapter: complete search, greedy, divide and conquer, and dynamic programming.
- Complete search (aka brute force) is a method for solving a problem by traversing the entire search space to find a solution. During the search, we can prune parts of the search space that we are sure do not lead to the required solution.
- A greedy algorithm takes a locally optimum choice at each step, with the hope of eventually reaching a globally optimum solution.
- Divide and conquer is a technique that divides a problem into smaller, independent subproblems and then combines solutions for each of the subproblems.
- Dynamic programming (DP) is a technique that divides a problem into smaller overlapping subproblems, computes a solution for each subproblem, and stores it in a DP table. The final solution is read off the DP table.
- Mastery of algorithms and software implementation can be achieved through the practice of competitive programming (see appendix A for resources).
- Machine learning mastery requires a solid understanding of fundamentals. See the recommended texts section in appendix A for ideas on how to increase the depth and breadth of your knowledge.
- The field of machine learning is rapidly evolving, and the best way to remain current is to read the latest research conference papers.