Algorithms

Muchang Bahng

Spring 2024

Contents

1 Complexity **2**
1.1 Recursive Algorithms and Recurrence Relation 3

2 Brute Force Algorithms **4**
2.1 Basic Arithmetic 4
2.2 Lists 5
2.3 Cryptography 5
2.4 Matrix Operations 5

3 Divide and Conquer **6**
3.1 Karatsuba Algorithm for Multiplication 6
3.2 Merge Sort 6
3.3 Strassen Algorithm 6
3.4 Fast Fourier Transform 6

4 Hashing and Probabilistic Algorithms **6**
4.1 Modulo Operations 6
4.2 Hashing 6
4.3 Primality Testing 6

5 Dynamic Programming **6**
5.1 Longest Increasing Subsequence 6
5.2 Knapsack 6

6 Graphs **6**
6.1 Representations and Properties 6
6.2 Exploration 8
6.3 Bipartite Graphs 9

7 Approximation Algorithms **10**
7.1 Greedy Algorithms 10

8 Linear Programming **10**

1 Complexity

A course on the study of algorithms.

Definition 1.1 (Algorithm)

An **algorithm** is a procedure for solving a mathematical problem in a *finite* number of steps. It should be

1. finite,
2. correct,
3. efficient

An algorithm, with respect to some inputs $\mathbf{n}$, will have a runtime that is some function f. We would like a formal way to analyze the asymptotic behavior between two functions.

Definition 1.2 (Complexity)

Given two positive functions f, g,

1. $f = O(g)$ if f/g is bounded.[a]
2. $f = \Omega(g)$ if g/f is bounded, i.e. $g = O(f)$.
3. $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$.

There are two notions of complexity here. We can compare f and g with respect to the *value* N of the input, or we can compare them with respect to the *number of bits* n in the input. While we mostly use the complexity w.r.t. the value, we should be aware for certain (especially low-level operations), the bit complexity is also important.

Let's do a quick comparison of various functions. Essentially, if we want to figure out the complexity of two positive functions f, g,[1] we can simply take the limit.

$$\lim_{x \to +\infty} \frac{f(x)}{g(x)} = \begin{cases} 0 & \implies f = O(g) \\ 0 < x < +\infty & \implies f = \Theta(g) \\ +\infty & \implies f = \Omega(g) \end{cases} \tag{1}$$

Most of the time, we will have to use L'Hopital's rule to derive these actual limits, but the general trend is

1. $\log n$ is small
2. $\text{poly}(n)$ grows faster
3. $\exp(n)$ grows even faster
4. $n!$ even faster
5. n^n even faster

Theorem 1.1 (Properties)

Some basic properties, which shows very similar properties to a vector space.

1. Transitivity.

$$f = O(g), g = O(h) \implies f = O(h) \tag{2}$$
$$f = \Omega(g), g = \Omega(h) \implies f = \Omega(h) \tag{3}$$
$$f = \Theta(g), g = \Theta(h) \implies f = \Theta(h) \tag{4}$$
$$\tag{5}$$

[a]Note that it is more accurate to write $f \in O(g)$, since we consider $O(g)$ a class of functions for which the property holds.
[1]These will be positive since the runtime must be positive.

2. Linearity.

$$f = O(h), g = O(h) \implies f + g = O(h) \tag{6}$$
$$f = \Omega(h), g = \Omega(h) \implies f + g = \Omega(h) \tag{7}$$
$$f = \Theta(h), g = \Theta(h) \implies f + g = \Theta(h) \tag{8}$$
$$\tag{9}$$

Example 1.1 ()

Compare the following functions.

1. $f(n) = \log_{10}(n), g(n) = \log_2(n)$. Since they are different bases, we can write $f(n) = \log(n)/\log(10)$ and $g(n) = \log(n)/\log(2)$. They differ by a constant factor, so $f = \Theta(g)$.
2. $f(n) = (\log n)^{20}, g(n) = n$. We have

$$\lim_{n\to\infty} \frac{(\log n)^2 0}{n} = \lim_{n\to\infty} \frac{20 \cdot (\log n)^{19} \cdot \frac{1}{n}}{1} = \ldots = \lim_{n\to\infty} \frac{20!}{n} = 0 \implies f = O(g) \tag{10}$$

3. $f(n) = n^{100}, g(n) = 1.01^n$. We have

$$\lim_{n\to\infty} \frac{n^{100}}{1.01^n} = \lim_{n\to\infty} \frac{100n^{99}}{1.01^n \cdot \log(1.01)} = \ldots = \lim_{n\to\infty} \frac{100!}{1.01^n \cdot (\log 1.01)^1 00} = 0 \implies f = O(g) \tag{11}$$

Let's do a slightly more nontrivial example.

Example 1.2 ()

Given the following algorithm, what is the runtime?

```
for i in range(1, n+1):
  j = 1
  while j <= i:
    j = 2 * j
```

Now we can see that for each i, we will double up to $\log_2(i)$ times. Therefore summing this all over i is

$$\sum_{i=1}^n \log_2(i) = \log_2(n!) \leq \log_2(n^n) = n \log_2(n) \tag{12}$$

and so we can see that the runtime is $O(n \log n)$. Other ways to do this is to just replace the summation with an integral.[a]

$$\int_1^n \log_2(x)\, dx = x \log(x) - x\Big|_1^n = n \log(n) - n + 1 = O(n \log n) \tag{13}$$

1.1 Recursive Algorithms and Recurrence Relation

I assume that the reader is familiar with recursive algorithms. Now to evaluate the runtime of a recursive algorithm, one must implicitly solve for the runtime of its recursive calls.

[a]Need more justification on why this is the case. Mentioned in lecture.

2 Brute Force Algorithms

2.1 Basic Arithmetic

In here, we use basic deductions from elementary algebra to give us a starting point at which we analyze fundamental arithmetic algorithms.

Theorem 2.1 (Complexity of Addition)

The complexity of addition of two $O(N)$ values with n bits is
1. $O(n)$ bit complexity.
2. $O(\log N)$ complexity.
3. $O(1)$ memory complexity.

By the same logic, the complexity of subtraction is
1. $O(n)$ bit complexity.
2. $O(\log N)$ complexity.
3. $O(1)$ memory complexity.

Proof.

To see bit complexity, we are really taking each bit of each number and adding them together, plus a potential carry operation. Therefore, we are doing a bounded number of computations per bit, which is $O(1)$, but we must at least read through all of the bits, making this $O(\max\{n, m\})$.

Theorem 2.2 (Complexity of Multiplication)

The complexity of multiplication of two values N, M with bits n, m is
1. $O(n^2)$ bit complexity.[a]
2. $O((\log n)^2)$ complexity.
3.

Theorem 2.3 (Complexity of Division)

The complexity of multiplication of two values N, M with bits n, m is
1. $O(n^2)$ bit complexity.
2.

Theorem 2.4 (Complexity of Modulus)

The complexity of multiplication of two values N, M with bits n, m is
1. $O(n^2)$ bit complexity.
2.

Theorem 2.5 (Complexity of Exponentiation)

The complexity of multiplication of two values N, M with bits n, m is
1. $O(n^2)$ bit complexity.
2.

[a]It turns out we can do better, which we will learn later.

Theorem 2.6 (Complexity of Square Root)

The complexity of multiplication of two values N, M with bits n, m is

1. $O(n^2)$ bit complexity.
2.

Definition 2.1 (Factorial)

2.2 Lists

Definition 2.2 (Max and Min of List)

Definition 2.3 (Bubble Sort)

Definition 2.4 (Binary Search)

2.3 Cryptography

Example 2.1 (GCD of Two Numbers)

Take a look at the following algorithm.

```
def gcd(a, b):
  if a == b:
    return a
  elif a > b:
    return gcd(a - b, b)
  else:
    return gcd(a, b - a)

print(gcd(63, 210))
```

Definition 2.5 (Primality Testing)

Definition 2.6 (Integer Factorization)

2.4 Matrix Operations

Definition 2.7 (Matrix Multiplication)

Definition 2.8 (Singular Value Decomposition)

Definition 2.9 (QR Decomposition)

Definition 2.10 (LU Decomposition)

Definition 2.11 (Matrix Inversion)

3 Divide and Conquer

Definition 3.1 (Divide and Conquer Algorithms)

3.1 Karatsuba Algorithm for Multiplication

3.2 Merge Sort

3.3 Strassen Algorithm

3.4 Fast Fourier Transform

4 Hashing and Probabilistic Algorithms

4.1 Modulo Operations

4.2 Hashing

4.3 Primality Testing

5 Dynamic Programming

5.1 Longest Increasing Subsequence

5.2 Knapsack

6 Graphs

A huge portion of problems can be solved by representing as a *graph* data structure. In here, we will explore various problems that can be solved through *graph algorithms*.

6.1 Representations and Properties

All graphs consist of a set of vertices/nodes V and edges E. This tuple is what makes up a graph. We denote $|V| = n, |E| = m$.

Definition 6.1 (Undirected Graphs)

An **undirected graph** $G(V, E)$ is a tuple, where $V = \{v_1, \ldots, v_n\}$ is the vertex set and $E = \{\{v_i, v_j\}\}$ is the edge set (note that it is a set of sets!).

1. The **degree** d_v of a vertex v is the number of edges incident to it.
2. A **path** is a sequence of vertices where adjacent vertices are connected by a path in E. It's **length** is the number of edges in the path.
3. A **cycle** is a path that has the same start and end.
4. A graph is **connected** if for every pair of vertices $e_i, e_j \in E$, there is a path from
5. A **connected component** is a maximal subset of connected vertices.

Definition 6.2 (Directed Graph)

A **directed graph** $G(V, E)$ is a tuple, where $V = \{v_1, \ldots, v_n\}$ is the vertex set and $E = \{(v_i, v_j)\}$ is the edge set (note that it is a set of tuples, so $(i, j) \neq (j, i)$).

1. The **in/out degree** $d_{v,i}, d_{v,o}$ of a vertex v is the number of edges going in to or out from v.
2.

In fact, from these definitions alone, we can solve an ancient puzzle called *the Bridges of Konigsberg*. Euler, in trying to solve this problem, had invented graph theory.

Example 6.1 (Bridges of Konigsberg)

Is there a way to walk that crosses each bridge *exactly* once?

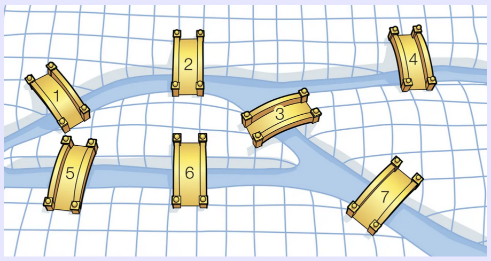

Figure 1: Bridges of Konigsberg

It can be decomposed into this undirected graph.

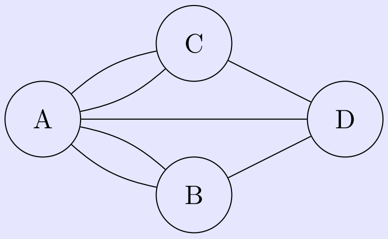

Figure 2: Graph representation.

Euler's observation is that except for start and end points, a talk leaves any vertex by different edge that the incoming edge. Therefore, the degree (number of edges incident on it) must have an even number, so all but 2 vertices must have an even degree. Since every vertex has an odd degree, there is no way of doing it.

In addition to the *adjacency list* representation, another way in which we represent a directed graph is through *adjacency matrices*.

Definition 6.3 (Adjacency Matrix)

In a finite directed graph (V, E), we can construct a bijection from V to the natural numbers and so we label each element in V with $i \in \mathbb{N}$. Then, we can construct a matrix A such that

$$A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{if } (i,j) \not\in E \end{cases} \tag{14}$$

While the adjacency matrix does have its advantages and has a cleaner form, usually in sparse graphs this is memory inefficient due to there being an overwhelming number of 0s.

Definition 6.4 (Trees)

An undirected graph $G(V, E)$ is a **tree** if

1. G is connected.
2. G has no cycles.[a]

Removing the first requirement gives us the definition of a **forest**, which is a collection of trees. Conversely, if $G(V, E)$ is connected and $|E| = n - 1$, then G is a tree.

Theorem 6.1 (Properties of Trees)

If $G(V, E)$ is a tree, then

1. There exists a $v \in V$ s.t. $d_v = 1$, called a **leaf node**.
2. $|E| = |V| - 1 = n - 1$.

Proof.

The outlines are quite intuitive.

1. There must be some leaf node since if there wasn't, then we would have a cycle. We can use proof by contradiction.
2. We can use proof by induction. We start off with one vertex and to construct a tree, we must add one edge and one vertex at every step, keeping this invariant.

6.2 Exploration

Given two $v, s \in V$ either directed or undirected, how can we find the shortest path from v to s?

Definition 6.5 (Depth First Search)

Theorem 6.2 (Runtime of DFS)

Definition 6.6 (Breath First Search)

This can be used to make a *BFS tree*.

[a]This makes sense, since to get back to a previous vertex you must backtrack.

Definition 6.7 (BFS Tree)

Theorem 6.3 (Runtime of BFS)

To get the running time, we know that each vertex is popped only once from the queue, giving us $O(n)$. For each pop, we are exploring all the neighbors of V.

$$\begin{aligned} O\bigg(\sum_{v \in V} |\text{neighbors of } v| + 1\bigg) &= O\bigg(\sum_{v \in V} d_v + 1\bigg) && (15) \\ &= O(2|E| + |V|) = O(m+n) && (16) \end{aligned}$$

which is linear in input size!

Exercise 6.1 ()

Prove that in any connected undirected graph $G = (V, E)$ there is a vertex $v \in V$ s.t. G remains connected after removing v.

Proof.

You can make a BFS tree from this graph, and there has to be a last layer. Just remove the node from this layer.

Exercise 6.2 ()

Give an example of a strongly connected directed graph $G = (V, E)$ s.t. that every $v \in V$, removing v from G gives a directed graph that is not strongly connected.

Exercise 6.3 (DPV 3.16)

Proof.

Do BFS but now recursively store the maximum of all the prerequisite courses plus 1 for the current course.

6.3 Bipartite Graphs

Definition 6.8 (Bipartite Graph)

A **bipartite graph** is an undirected graph $G(V, L)$ where we can partition $V = L \sqcup R$ such that for all $e = \{u, v\} \in E$, we have $u \in L, v \in R$.

We would like to devise some method to determine if an arbitrary graph is bipartite.

Theorem 6.4 ()

G is bipartite if and only if all cycles in G are even length.

Proof.

Proving ($\implies$) is quite easy since if we suppose G has an odd length cycle, then we start packing vertices of a cycle into L, R, but by the time we came back to the start, we are forced to pack it into the wrong partition!
The converse is quite hard to prove, and we'll take it at face value.

Example 6.2 (Determine Bipartite)

Given an arbitrary graph, I want to determine if it is bipartite.

1. We first run BFS on the graph starting at $s \in V$, which divides it up into layers representing the shortest path from s.
2. Then for each layer $L_i \subset V$, we check if there are connections between two vertices $x, y \in L_i$. If there are connections, then this is not bipartite. If there are none, then this is bipartite.

7 Approximation Algorithms

7.1 Greedy Algorithms

8 Linear Programming