Discrete Math Algorithms Writeup

Nashir Janmohamed, Ariel Young

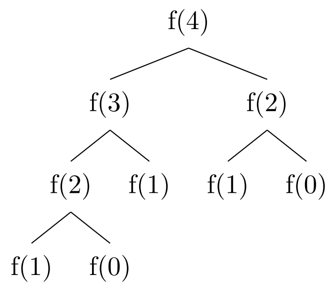
June 14th, 2020

Summary

1 Recurrence Relations/Dynamic Programming

Recursive algorithms make an exponential number of iterations to solve a problem. The *recursion tree* (a diagram of the tree of recursive calls) (1) grows exponentially with n and does not take advantage of previously solved subproblems. For example when $n = 4$, the computation for $f(1)$ is performed three times.

Figure 1: Fibonacci recursion tree for n = 4



A problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems (2). Recursively defined problems oftentimes have this structure, and an alternative technique called dynamic programming can be used to more efficiently solve them.

Consider the following example (3) using the Fibonacci numbers (see Figure Fibonacci numbers for an explanation of the Fibonacci numbers) for an intuition on how dynamic programming can more efficiently solve a problem with a recursive solution.

A naive recursive implementation to compute the nth Fibonacci sequence is as follows

```
def fib(x):
    if (x < 2):
        return 1

    return fib(x-1) + fib(x-2)
```

At each step, 2 recursive calls are made, so the time complexity is $O(2^n)$ (exponential). There are also two calls placed on the stack at each step, so the space complexity is also $O(2^n)$.

The following dynamic programming implementation is iterative (no recursive calls are made), and computes the nth fibonacci in linear time, $O(n)$, by building the result from the bottom up. This algorithm also uses $O(n)$ extra space to hold the result of previous computations (which with modification to store fibresult after function results, could be useful if the function is called more than once).

```
def fib(n):
    fibresult[0] = 1
    fibresult[1] = 1
    for i in range(2, n):
        fibresult.append(fibresult[i-1] + fibresult[i-2])
    return fibresult[n]
```

The most efficient implementation is O(n) time and constant space

```
def fib(int n):
    a = 1
    b = 1
    for i in range(2, n+1):
        a = a + b;
        a, b = b, a # swap variables
    return b
```

For each of the algorithms implemented, a recursive and dynamic programming implementation was created.

1.1 Bell numbers

The Bell numbers represent the number of ways to count partitions of (or equivalently equivalence relations on) an n element set. The n-th bell number is given by the recurrence

$$B_n = \sum_{k=1}^{n} \binom{n-1}{k-1} B_{n-k}$$

for $n \geq 0$.

1.2 Catalan numbers

The Catalan numbers form a sequence of natural numbers that occur in various counting problems, often involving recursively-defined objects. They can be expressed by the recurrence relation

$$C_{n+1} = \sum_{i=0}^{n} C_i C_{n-1}$$

for $n \geq 0$.

1.3 Fibonacci numbers

The Fibonacci numbers, commonly denoted F_n, form a sequence such that each number is the sum of the two preceding ones, with $F_0 = 0$, $F_1 = 1$, and the recurrence given by

$$F_n = F_{n-1} + F_{n-2}$$

for $n > 1$.

1.4 Solve linear homogeneous recurrence relations with constant coefficients (LHCCRRs)

A linear homogeneous recurrence relation with constant coefficients (LHCCRR) of degree k is a recurrence relation of the form

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$$

where c_1, c_2, …, $c_k \in R$ and $c_k \neq 0$. The *characteristic equation* (1) is

$$r_k - c_1 r_{k-1} - c_2 r_{k-2} - \ldots c_k = 0$$

Suppose that (1) has t distinct roots $r_1, r_2, \ldots, r_t$ with multiplicities $m_1, m_2, \ldots, m_t$, respectively, so that $m_i \geq 1$ for $i = 1, 2, \ldots, t$ and $m_1 + m_2 + \cdots + m_t = k$. Then a sequence a_n is a solution of the recurrence relation if and only if

$$\begin{aligned} a_n =& (\alpha_{1,0} + \alpha_{1,1} n + \cdots + \alpha_{1,m_1-1} n^{m_1-1}) r_1^n \\ &+ (\alpha_{2,0} + \alpha_{2,1} n + \cdots + \alpha_{2,m_2-1} n^{m_2-1}) r_2^n \\ &+ \cdots + (\alpha_{t,0} + \alpha_{t,1} n + \cdots + \alpha_{t,m_t-1} n^{m_t-1}) r_t^n \end{aligned}$$

for $n = 0, 1, 2, \ldots$, where $\alpha i, j$ are constants for $1 \leq i \leq t$ and $0 \leq j \leq m_i - 1$. (This implementation only solves recurrence relations with non-complex roots.)

2 Permutations and Combinations

2.1 Combinations without repetition

A combination without repetition is a selection of items from a collection, such that the order of selection does not matter. A k-combination of an n element set S is a subset of k distinct elements. The number of k-combinations is equal to the binomial coefficient given by

$$\binom{n}{k} = \frac{n!}{(n-k)!\,k!}$$

2.2 Permutations without repetition

k-permutations of n are the different ordered arrangements of a k-element subset of an n-set. This number is given by

$$P(n,k) = n \cdot (n-1) \cdot (n-2) \cdot \ldots (n-k+1) = \frac{n!}{(n-k)!}$$

2.3 Combinations without repetition

A k-combination with repetitions allowed is a sequence of k not necessarily distinct elements of S, where order is not taken into account, i.e. the number of ways to sample k elements from a set of n elements allowing for duplicates but disregarding different orderings. Using the *stars and bars method*, it can be shown that this number is given by

$$\binom{n+k-1}{k}$$

2.4 Permutations with repetition

Permutations with repetition are ordered arrangements of k elements from a set S with n elements where repetition is allowed. The number of permutations with repetition of size k is simply k^n (except if $k > n$, where the result is 1).

2.5 Generate permutations of a string

Given a string s of length n, generate all $n!$ permutations of s. For example, all the permutations of "the" are ["the", "teh", "het", "hte", "eth", "eht"].

2.6 Generate all bit strings of length n

Given an integer n, all bit strings of length n can be recursively constructed by appending a 0 and a 1 to all bit strings of length $n - 1$.
Thus, there are $2 * (s_{n-1})$ bit strings of length n. Since there are 2 bit strings of length 1, there are 2^n bit strings of length n.

3 Relations

3.1 # of relations

A relation on an n element set S is a subset of $S \times S$, or equivalently, an element of the power set of $S \times S$. There are

$$2^{|S||S|} = 2^{n^2}$$

such subsets.

3.2 # of transitive relations

There is no known closed formula for counting the number of transitive relations. The (perhaps inefficient) approach taken in this algorithm is as follows

(1) Generate all possible relations for an n element set (given by the power set of the cartesian product of the set $\{1, 2, 3, \ldots, n\}$)

(2) For each relation generated in (1), check that for each (a, b), if there is a point of the form (b, c), then (a, c) must be in the relation

3.3 # of (ir)reflexive relations

A relation is reflexive if all elements are related to themselves, or equivalently, all entries on the main diagonal of the matrix representation of the relation must be 1. There are n^2 entries in the matrix and n entries on the main diagonal.

For the remaining $n^2 - n$ off diagonal entries, the ordered pair may or may not be in the relation. Thus, there are

$$2^{n^2-n}$$

reflexive relations. The argument for irreflexive relations is the same, with the exception that all entries on the main diagonal of the matrix representation of the relation must be 0.

3.4 # of symmetric relations

A relation R is symmetric if for all (a, b) that are in R, (b, a) is also in R. Each element on the diagonal may or may not be related to itself, and similarly for all the $\binom{n}{2}$ two element subsets (with distinct elements). Thus, there are

$$2^{\binom{n}{2}+n} = 2^{\frac{n(n+1)}{2}}$$

symmetric relations on a set with n elements.

3.5 # of antisymmetric relations

A relation R is antisymmetric if for all (a, b) that are in R, if (b, a) is in R, then $a = b$. There are two choices for every element on the diagonal. For the remaining $\binom{n}{2}$ two element subsets (with distinct elements) with elements a and b, either $(a, b) \in R$ and $(b, a) \notin R$, $(a, b) \notin R$ and $(b, a) \in R$, or $(a, b) \notin R$ and $(b, a) \notin R$, so there are 3 choices for each two element subset. Thus, there are

$$2^n 3^{\binom{n}{2}} = 2^n 3^{\frac{n(n-1)}{2}}$$

antisymmetric relations on a set with n elements.

3.6 # of equivalence relations

A relation R on a set A is an equivalence relation if it is reflexive, symmetric, and transitive. For each $a \in A$, the equivalence class of a is given by $[a] = \{x \mid xRa\}$. The equivalence classes form a partition of A, and so the number of equivalence relations on a set S is given by the number of partitions of a set S. So, this is just given by the Bell numbers [see section 1].

$$B_n = \sum_{k=1}^{n} \binom{n-1}{k-1} B_{n-k}$$

for $n \geq 0$.

4 Sets

4.1 Generate power set

The power set of a set S is the set of all subsets of S, denoted by $\mathcal{P}(S)$. For each element in S, it either is or is not in a subset of S, so there are two choices for each element of S in each subset. Thus, there are $2^{|S|}$ subsets of S. Alternatively, the inclusion of an element in a subset can be represented by a bitstring of length $|S|$, with a 1 indicating the element is in the subset and a 0 indicating otherwise. This was used as a baseline for the algorithm - all bit strings of length n were generated, and if the ith bit was set, the ith element was included in the subset.

4.2 Generate cartesian product

The Cartesian product of two sets A and B, denoted $A \times B$, is the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$. In terms of set-builder notation, that is

$$A \times B = \{\, (a, b) \mid a \in A \ \text{ and } \ b \in B \,\}$$

This function can be called with either one or two sets. If one set A is passed, then the result will be $A \times A$. If two sets are passed, separate them with a semicolon, i.e. "1, 2, 3; 4, 5".

5 References

References

[1] N. Foster, "Lecture 20, cs3110 spring 2012, data structures and functional programming," April 2012.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd ed., 2009.

[3] T. (https://stackoverflow.com/users/1131467/andrew tomazos), "What's the difference between recursion, memoization & dynamic programming?."