Algorithmic Journeys

Generic algorithms and performance

Taras Shevhcnkeo

Rails Reactor

Table of contents

Terminology

Terminology

1. Datum
2. Value
3. Value type
4. Object
5. Object type

Datum

Definition

A **datum** is a sequence of bits.

Example

01000001 is an example of a datum.

Value

Definition

A **value is** a **datum** together with its interpretation.

Example

The **datum** 01000001 might have the interpretation of the integer 65, or the character “A”.

Explanation

Every **value** must be associated with a **datum** in memory; there is no way to refer to disembodied **values** in modern programming languages.

Value type

Definition

A **value type** is a set of values sharing a common interpretation.

Object

Definition

An **object** is a collection of bits in memory that contain a **value** of a given **value type**.

Explanation

An **object** is immutable if the value never changes, and mutable otherwise. An object is unrestricted if it can contain any **value** of its **value type**.

Object type

Definition

An **object type** is a uniform method of storing and retrieving **values** of a given **value type** from a particular **object** when given its address.

Programming with concepts

Basic idea

The essence of generic programming lies in the idea of concepts. A concept is a way of describing a family of related object types.

Natural Science	Mathematics	Programming	Programming Examples
genus	theory	concept	Integral, Character
species	model	type or class	uint8_t, char
invidiual	element	instance	01000001(65, 'A')

Notion of Regularity

Operation

1. Copy construction
2. Assignment
3. Equality
4. Destruction

Semantic

$$\forall a\ \forall b\ \forall c : T\ a(b) \implies (b = c \implies a = c)$$

$$\forall a\ \forall b\ \forall c : a \leftarrow b \implies (b = c \implies a = c)$$

$$\forall f \in RegularFunction : a = b \implies f(a) = f(b)$$

More examples of concepts

1. Regular Type
2. Semiegular Type
3. Functional Procedure
4. Homogeneous Function
5. Homogeneous Predicate
6. Semiring
7. Sequence
8. Totally Ordered
9. Input Iterator
10. Forfward Iterator
11. Bidirectional Iterator

Properties

1. Associative
2. Distributive
3. Transitive
4. Semiegular Type
5. Functional Procedure

Techniques

1. Transformation-action duality
2. Operation-accumulation procedure duality
3. Memory adaptivity
4. Reduction to constrained subproblem

Egyptian multiplication

Simple algorithm

3 * 8

x	y
3	8
6	7
9	6
12	5
15	4
18	3
21	2
24	1

8 * 3

x	y
8	3
16	2
24	1

Far away from egyptian multiplication

Code

```
def intersect(x, y):
    if len(x) > len(y): x, y = y, x
    for i, v in enumerate(x):
        if v in y: yield v; print(i);

x, y = set([1, 2]), set(range(10**7))
print(set(intersect(x, y)))
```

Output

```
0
1
{1, 2}
```

The first implementation

Code

```
template<typename T, typename N>
T multiply0(T x, N n) {
  if (n == 1) return x;
  return multiply0(x, --n) + x;
}
```

Egyptian multiplication - 1

Code

```
template<typename T, typename N>
T multiply_accum1(T x, N n, T r) {
  if (n == 1) return x + r;
  return multiply_accum1(x, n - 1, r + x);
  return x;
}

template<typename T, typename N>
T multiply1(T x, N n) {
  return multiply_accum1(x, n, T(0));
}
```

Egyptian multiplication - 2

Code

```
template<typename T, typename N>
T multiply_accum2(T x, N n, T r) {
  if (n == 1) return x + r;
  --n;
  r = r + x;
  return multiply_accum2(x, n, r);
}

template<typename T, typename N>
T multiply2(T x, N n) {
  return multiply_accum2(x, n, T{0});
}
```

Egyptian multiplication - 3

Code

```
template<typename T, typename N>
T multiply_accum3(T x, N n, T r) {
  while(true) {
    if (n == 1) return x + r;
    --n;
    r = r + x;
  }
}

template<typename T, typename N>
T multiply3(T x, N n) {
  return multiply_accum3(x, n, T{0});
}
```

Code

```
template<typename T, typename N>
T multiply4(T x, N n) {
  if (n == 1) return x;
  T r = multiply4<T, N>(x + x, half(n));
  if (odd(n)) r = r + x;
  return r;
}
```

Egyptian multiplication - 5

Code

```
template<typename T, typename N>
T multiply_accum5(T x, N n, T r) {
  if (n == 1) return r + x;
  if (odd(n)) {
    return multiply_accum5<T, N>(x + x, half(n), r + x);
  } else {
    return multiply_accum5<T, N>(x + x, half(n), r);
  }
}

template<typename T, typename N>
T multiply5(T x, N n) {
  return multiply_accum5<T, N>(x, n, T{0});
}
```

Egyptian multiplication - 6

Code

```
template<typename T, typename N>
T multiply_accum6(T x, N n, T r) {
  if (n == 1) return r + x;
  if (odd(n)) r = r + x;
  n = half(n);
  x = x + x;
  return multiply_accum6(x, n, r);
}
template<typename T, typename N>
T multiply6(T x, N n) {
  return multiply_accum6(x, n, T{0});
}
```

Egyptian multiplication - 7

Code

```
template<typename T, typename N>
T multiply_accum7(T x, N n, T r) {
  while(true) {
    if (n == 1) return r + x;
    if (odd(n)) r += x;
    x += x;
    n = half(n);
  }
}
template<typename T, typename N>
T multiply7(T x, N n) {
  return multiply_accum7(x, n, T{0});
}
```

Code

```
template<typename T, typename N>
T multiply_accum8(T x, N n, T r) {
  while(true) {
    if (odd(n)) {
      r += x;
      if (n == 1) return r;
    }
    x += x;
    n = half(n);
  }
}
template<typename T, typename N>
T multiply8(T x, N n) {
  return multiply_accum8(x, n, T{0});
}
```

Egyptian multiplication - 9

Code

```
template<typename T, typename N>
T multiply9(T x, N n) {
  if (n == 1) return x;
  --n;
  return multiply_accum8(x, n, x);
}
```

Egyptian multiplication - 10

Code

```
template<typename T, typename N>
T multiply10(T x, N n) {
  while(!odd(n)) {
    x += x;
    n = half(n);
  }
  if (n == 1) return x;
  --n;
  return multiply_accum8(x, n, x);
}
```

Egyptian multiplication - Generic version

Code

```
template<typename T, typename N, typename Op>
requires(Regular(T) && Integer(N) &&
         SemigroupOperation(Op) && Domain<T, Op>)
T power_accumulate(T x, N n, T r, Op op) {
  while(true) {
    if (odd(n)) {
      r = op(r, x);
      if (n == 1) return r;
    }
    x = op(x, x);
    n = half(n);
  }
}
```

Egyptian multiplication - Generic version

Code

```
template<typename T, typename N, typename Op>
requires(Regular(T) && Integer(N) &&
        SemigroupOperation(Op) && Domain<T, Op>)
T power(T x, N n, Op op) {
  while(!odd(n)) {
    x = op(x, x);
    n = half(n);
  }
  if (n == 1) return x;
  return power_accumulate(op(x, x), half(n - 1), x, op);
}
```

Applications of Egyptian Multiplication

1. Multiplication
2. Pow
3. Transitive closure
4. Shortest path

Multiplication

1. Multiplication
2. Pow
3. Transitive closure
4. Shortest path

Egyptian Multiplication for Multiplication

Code

```
power_monoid(10, 30, std::plus<int>{});
```

Egyptian Multiplication for Pow

Code

```
power_monoid(2, 20, std::multiplies<int>{});
```

Code

```
power_monoid(2, 20, std::multiplies<int>{});
```

Egyptian Multiplication for Transitive Closure

Code

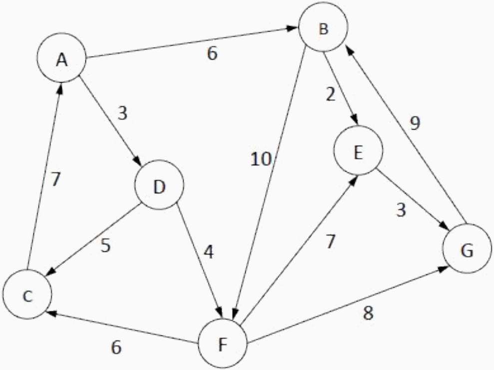
```
template<typename T> struct matrix_transitive_closure {
  matrix<T> operator()(const matrix<T> &a,
                       const matrix<T> &b) {
    auto closure = a;
    auto n = a.rows();
    for (size_t i = 0; i < n; ++i) {
      for (size_t j = 0; j < n; ++j) {
        T result = 0;
        for (size_t k = 0; k < n; ++k) {
          result = result | (a(i, k) & b(k, j)); }
        closure(i, j) = result; }}
    return b;}};
power_monoid(10, 30, matrix_transitive_closure<int>{});
```

Egyptian Multiplication for Shortest path

Code

```
template<typename T> struct tropical_semiring {
  matrix<T> operator()(const matrix<T> &a,
                       const matrix<T> &b) {
    auto closure = a;
    auto n = a.rows();
    for (size_t i = 0; i < n; ++i) {
      for (size_t j = 0; j < n; ++j) {
        T result = std::numeric_limits<T>::max();
        for (size_t k = 0; k < n; ++k) {
          result = min(result, a(i, k) + b(k, j)); }
        closure(i, j) = result; }}
    return b;}};
power_monoid(10, 30, matrix_transitive_closure<int>{});
```

Graph



Graph

$$\begin{bmatrix} 0 & 6 & inf & 3 & inf & inf & inf \\ inf & 0 & inf & inf & 2 & 10 & inf \\ 7 & inf & 0 & inf & inf & inf & inf \\ inf & inf & 5 & 0 & inf & 4 & inf \\ inf & inf & inf & inf & 0 & inf & 3 \\ inf & inf & 6 & inf & 7 & 0 & 8 \\ inf & 9 & inf & inf & inf & inf & 0 \end{bmatrix}$$

Shortest distance

$$\begin{bmatrix} 0 & 6 & 8 & 3 & 8 & 7 & 11 \\ 23 & 0 & 16 & 26 & 2 & 10 & 5 \\ 7 & 13 & 0 & 10 & 15 & 14 & 18 \\ 12 & 18 & 5 & 0 & 11 & 4 & 12 \\ inf & 12 & 28 & inf & 0 & 22 & 3 \\ 13 & 17 & 6 & 16 & 7 & 0 & 8 \\ 32 & 9 & 25 & inf & 11 & 19 & 0 \end{bmatrix}$$

Homework

1. Rewrite functors as algorithms
2. Play around linear recurrences

Conclusion

Conclusion

1. Concreteness costs
2. Abstracting algorithms to their most general setting without losing efficiency
3. Know your algorithms