Slowing Down to be Faster

C++ and Divisible Algorithms in Real-Time Systems

Patrice Roy – Patrice.Roy@USherbrooke.ca, Patrice.Roy@clg.qc.ca

CppNorth, July 2022

Suppose the following…

Suppose the following…

```
// ...
constexpr auto images_second = 60.0;
constexpr auto time_per_image = 1s/images_second;
while(!done) {
   const auto pre = high_resolution_clock::now();
   auto img = prepare_image();
   display(img);
   const auto post = high_resolution_clock::now();
   const auto remaining = time_per_image - (post - pre);
   this_thread::sleep_for(remaining);
}
// ...
```

Suppose the following…

```
// ... (simplified)
while(!done) {
    auto img = prepare_image();
    display(img);
    sleep_for(remaining_time);
}
// ...
```

Suppose the following…

```
// ... (simplified)
auto img = prepare_image();
while(!done) {
   display(img);
   img = prepare_image();
   sleep_for(remaining_time);
}
// ...
```

Suppose the following…

- Real-time systems are systems that never go too slow
- They typically guarantee that some constraints will be respected
  - [C] Constant iteration rate
  - [R] Regular iteration rate
  - [I] Immediate (low-latency; quick to start)
  - [B] Brief (quick to stop)

Suppose the following…

- Real-time systems are systems that never go too slow
- They typically guarantee that some constraints will be respected
    - **[C] Constant iteration rate**
    - [R] Regular iteration rate
    - [I] Immediate (low-latency; quick to start)
    - [B] Brief (quick to stop)

This is what concerns us today

Suppose the following…

```
// ... (simplified)
auto img = prepare_image();
while(!done) {
   display(img);
   img = prepare_image();
   sleep_for(remaining_time);
}
// ...
```

Suppose the following…

```
// ... (simplified)
auto img = prepare_image();
while(!done) {
    display(img);
    img = prepare_image();
    sleep_for(remaining_time);
}
// ...
```

We're wasting precious time here by sleeping, time we could use to make our system better… if we had something to do during that time!

What could we do there?

What could we do there?

- Many things, really
  - Decompress images
  - Prepare audio bits
  - Prefetch soon-to-be-used assets
  - etc.
- If there’s nothing to do, it’s can be reasonable to sleep
  - However…
- If there’s stuff to do, doing it now will make the overall system better
  - Less awkward pauses
  - Better fluidity

What could we do there?

```
// ... (simplified)
auto img = prepare_image();
while(!done) {
   display(img);
   img = prepare_image();
   sleep_for(remaining_time);
}
// ...
```

What could we do there?

```
// ...
constexpr auto images_second = 60.0;
constexpr auto time_per_image = 1s/images_second;
auto img = prepare_image();
while(!done) {
   const auto pre = high_resolution_clock::now();
   display(img);
   auto img = prepare_image();
   const auto post = high_resolution_clock::now();
   const auto remaining = time_per_image - (post - pre);
   this_thread::sleep_for(remaining);
}
// ...
```

What could we do there?

```
// ...
constexpr auto images_second = 60.0;
constexpr auto time_per_image = 1s/images_second;
auto img = prepare_image();
while(!done) {
   const auto pre = high_resolution_cl
   display(img);
   auto img = prepare_image();
   const auto post = high_resolution_clo  ..now();
   const auto remaining = time_per_image - (post - pre);
   this_thread::sleep_for(remaining);
}
// ...
```

Whatever we decide to do here, we cannot let it execute for a duration greater than `remaining`

Let’s use that time

Let’s use that time

- In this talk, we want use the remaining time to perform auxiliary tasks
  - The goal is to save time later, and make the overall system more fluid

Let’s use that time

- In this talk, we want use the remaining time to perform auxiliary tasks
  - The goal is to save time later, and make the overall system more fluid
- For the sake of this example, this auxiliary task will be to perform RLE compression on a sequence of pixels
  - Our pixels will simply be 24 bit RGB values

RLE compression – an overview

RLE compression – an overview

- RLE stands for Run-Length Encoding
- A RLE compression algorithm turns a sequence of values into a sequence of (*how many,what*) pairs
  - If *what* can be represented as an integral, the result of this process can be expressed as a sequence of integrals with an even number of elements

RLE compression – an overview

- RLE stands for Run-Length Encoding
- A RLE compression algorithm turns a sequence of values into a sequence of (*how many,what*) pairs
  - If *what* can be represented as an integral, the result of this process can be expressed as a sequence of integrals with an even number of elements

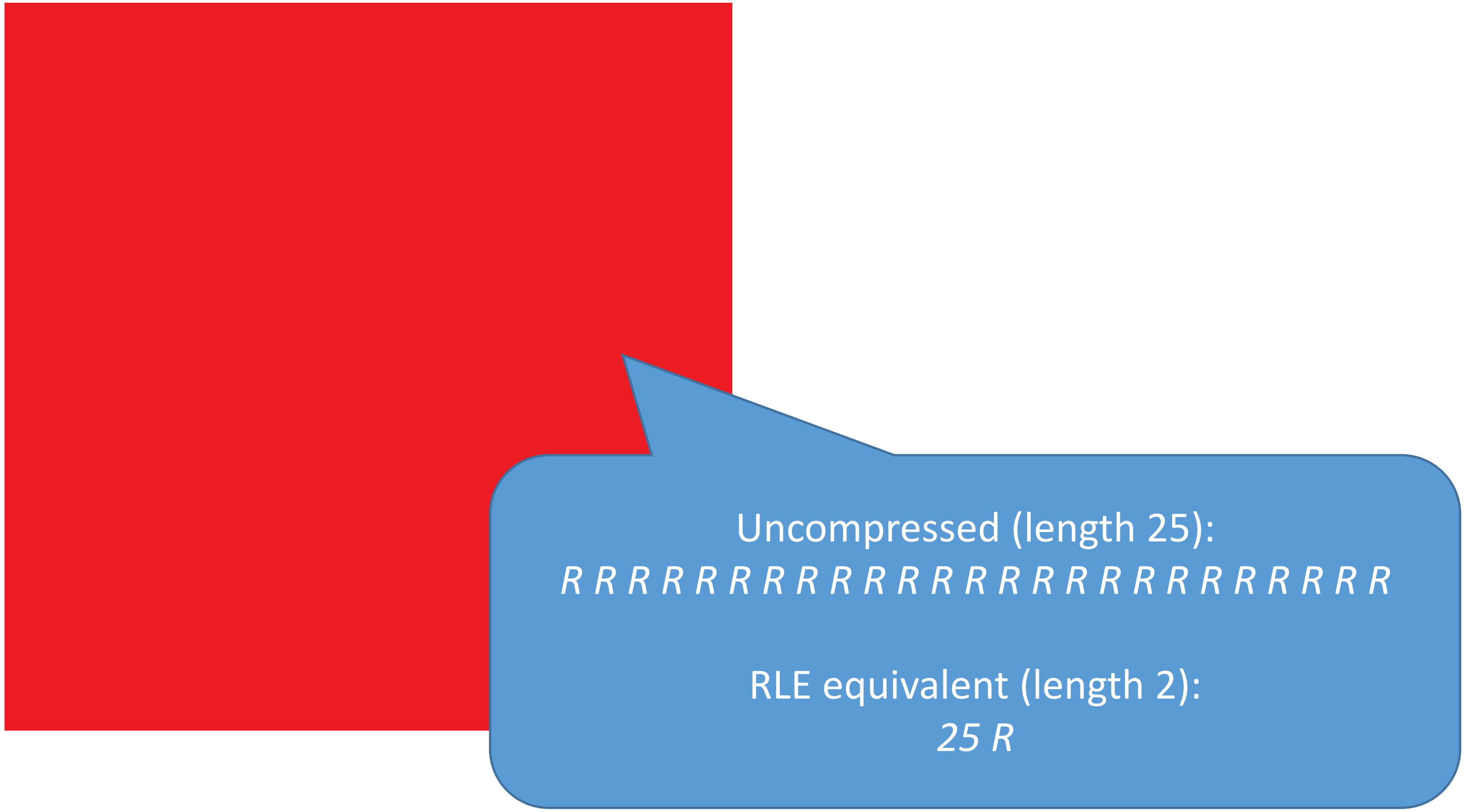
```
// before compression (length 21)
aaabbcccccdeeefffffgg
// after compression (length 14)
3a2b5c1d3e5f2g
```

RLE compression – an overview

RLE compression – an overview

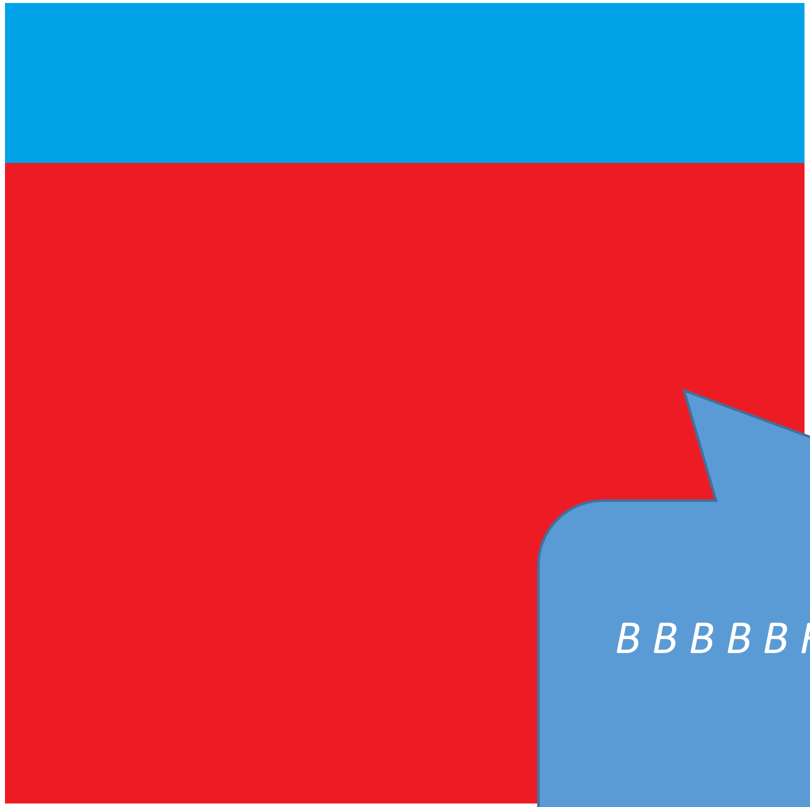
Suppose this is a 5x5 block of pixels. We'll use *R* for *red* and *B* for *blue*. Suppose the height, width, etc. metadata is stored elsewhere (we only care about the sequence of pixels)

RLE compression – an overview



RLE compression – an overview

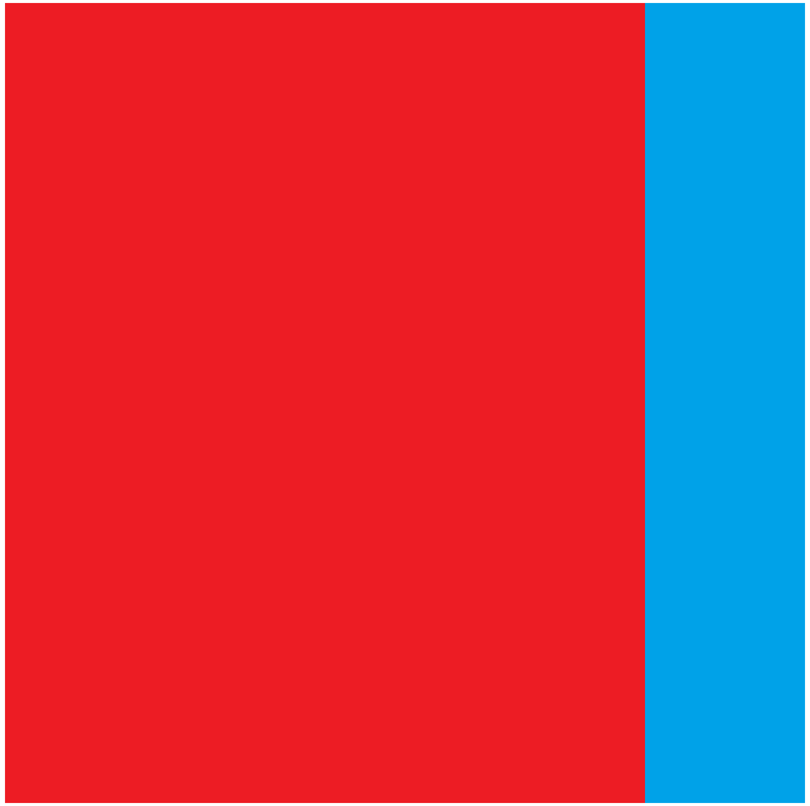
RLE compression – an overview



Uncompressed (length 25):
B B B B B R R R R R R R R R R R R R R R R R R R R

RLE equivalent (length 4):
5 B 20 R

RLE compression – an overview

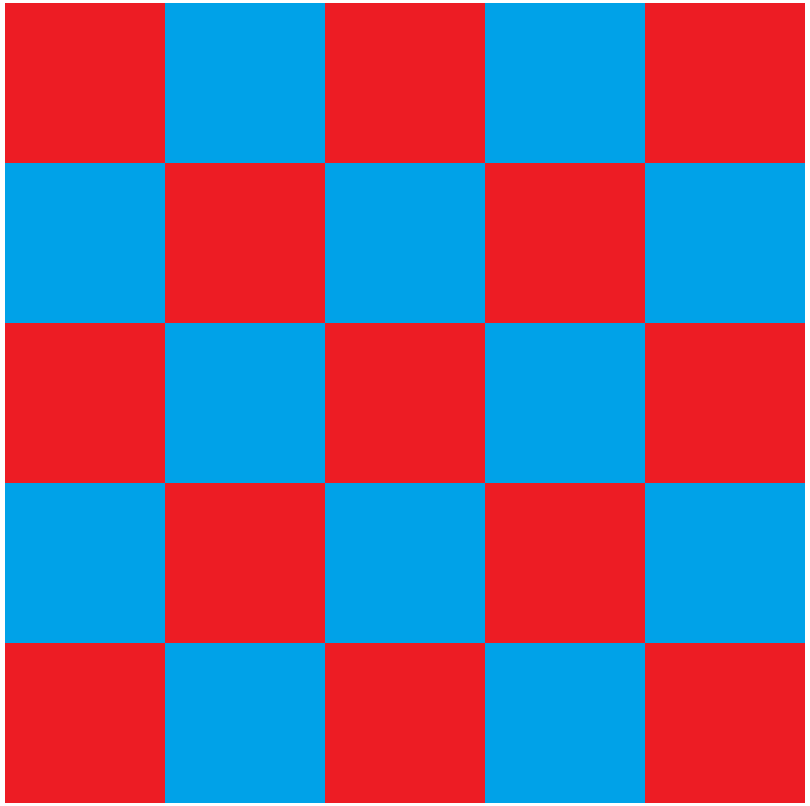


RLE compression – an overview

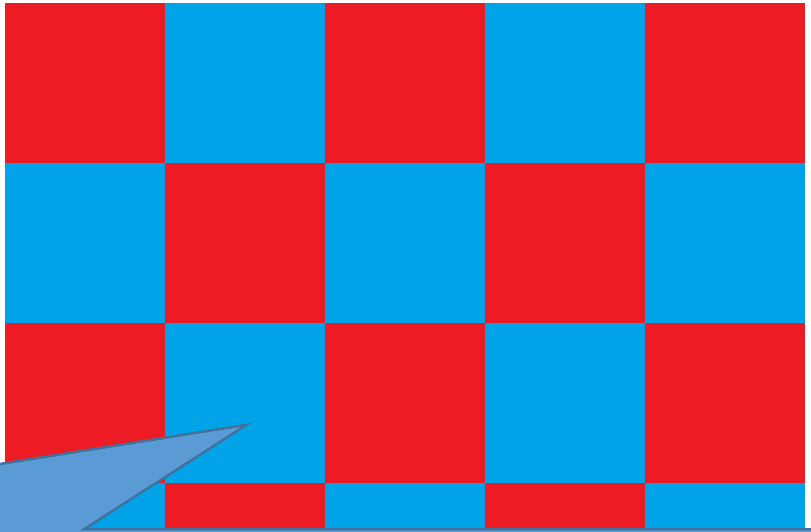
Uncompressed (length 25):
R R R R B R R R R B R R R R B R R R R B R R R R B

RLE equivalent (length 20):
4 R 1 B 4 R 1 B 4 R 1 B 4 R 1 B 4 R 1 B

RLE compression – an overview



RLE compression – an overview



Uncompressed (length 25):
R B R B R B R B R B R B R B R B R B R B R B R B R

RLE equivalent (length 50):
1 R 1 B 1 R 1 B 1 R 1 B 1 R 1 B 1 R 1 B 1 R 1 B 1 R 1 B 1 R 1 B 1 R 1 B 1 R 1 B 1 R 1 B 1 R 1 B 1 R

RLE compression – an overview

- Clearly, RLE is a simple algorithm, suitable for sequences with contiguous chunks of equivalent data
  - Cool for a child’s drawing, for example
  - Not cool for a photograph of a crowd in a city, or of a garden filled with flowers of varied colors

A note on data representation

A note on data representation

- In many problems, there is more than one way to represent data
- This one is not different
  - We can express a 24 bit RGB pixel in numerous ways
  - A non-exhaustive list follows

A note on data representation

- Encapsulation is beautiful
  - Design our algorithm based on the class' public interface
  - Empirically test data layouts to choose the best one
- Know your algorithms
  - It helps making informed choices of data representation

Designing a simple pixel class

Designing a simple pixel class

- There are many ways to design a pixel class for 24 bit RGB triples
  - Some examples of reasonable implementations follow

Designing a simple pixel class

```
// RGB, 24 bits : 8 bits red, 8 bits green, 8 bits blue
class pixel {
   unsigned char rgb[3]{};
public:
   pixel() = default;
   pixel(unsigned char r, unsigned char g, unsigned char b);
   // unsigned char red() const; // etc.
   bool operator==(const pixel &) const;
   // operator!= synthesized from operator== in C++20
   operator std::uint32_t() const;
};
```

Designing a simple pixel class

```
// RGB, 24 bits : 8 bits red, 8 bits green, 8 bits blue
class pixel {
   unsigned char rgb[3]{};
public:
   pixel() = default;
   pixel(unsigned char r, unsigned char g, unsigned char b);
   // unsigned char red() const; // etc.
   bool operator==(const pixel &) const;
   // operator!= synthesized from operator== in C++20
   operator std::uint32_t() const;
};
```

sizeof(pixel)==3
alignof(pixel)==1

Designing a simple pixel class

```
// RGB, 24 bits : 8 bits red, 8 bits green, 8 bits blue
class pixel {
   unsigned char rgba[4]{};
public:
   pixel() = default;
   pixel(unsigned char r, unsigned char g, unsigned char b);
   // unsigned char red() const; // etc.
   bool operator==(const pixel &) const;
   // operator!= synthesized from operator== in C++20
   operator std::uint32_t() const;
};
```

Designing a simple pixel class

```
// RGB, 24 bits : 8 bits red, 8 bits green, 8 bits blue
class pixel {
   unsigned char rgba[4]{};
public:
   pixel() = default;
   pixel(unsigned char r, unsigned char g, unsigned char b);
   // unsigned char red() const; // etc.
   bool operator==(const pixel &) const;
   // operator!= synthesized from operator== in C++20
   operator std::uint32_t() const;
};
```

sizeof(pixel)==4
alignof(pixel)==1

Designing a simple pixel class

```
// RGB, 24 bits : 8 bits red, 8 bits green, 8 bits blue
class pixel {
   unsigned char r{}, g{}, b{};
public:
   pixel() = default;
   pixel(unsigned char r, unsigned char g, unsigned char b);
   // unsigned char red() const; // etc.
   bool operator==(const pixel &) const;
   // operator!= synthesized from operator== in C++20
   operator std::uint32_t() const;
};
```

Designing a simple pixel class

```
// RGB, 24 bits : 8 bits red, 8 bits green, 8 bits blue
class pixel {
   unsigned char r{}, g{}, b{};
public:
   pixel() = default;
   pixel(unsigned char r, unsigned char g, unsigned char b);
   // unsigned char red() const; // etc.
   bool operator==(const pixel &) const;
   // operator!= synthesized from operator== in C++20
   operator std::uint32_t() const;
};
```

sizeof(pixel)==3
alignof(pixel)==1

Designing a simple pixel class

```
// RGB, 24 bits : 8 bits red, 8 bits green, 8 bits blue
class pixel {
   std::uint32_t rgba{};
public:
   pixel() = default;
   pixel(unsigned char r, unsigned char g, unsigned char b);
   // unsigned char red() const; // etc.
   bool operator==(const pixel &) const;
   // operator!= synthesized from operator== in C++20
   operator std::uint32_t() const;
};
```

Designing a simple pixel class

```
// RGB, 24 bits : 8 bits red, 8 bits green, 8 bits blue
class pixel {
   std::uint32_t rgba{};
public:
   pixel() = default;
   pixel(unsigned char r, unsigned char g, unsigned char b);
   // unsigned char red() const; // etc.
   bool operator==(const pixel &) const;
   // operator!= synthesized from operator== in C++20
   operator std::uint32_t() const;
};
```

sizeof(pixel)==4
alignof(pixel)==4

Designing a simple pixel class

- What should guide our choices?
  - Size of a pixel object
  - Risks of a pixel object falling on two cache lines
  - Most frequently used operations

Designing a simple pixel class

- What should guide our choices?
  - Size of a pixel object
  - Risks of a pixel object falling on two cache lines
  - **Most frequently used operations**

In our case, this is probably the most important factor to consider

There are many cases where differences in size and alignment impact Cache access significantly, and where the other aspects count much more. *Measure*…

Implementing RLE compression

Implementing RLE compression

```
vector<uint32_t> rle_compression(const vector<pixel> &pix) {
   vector<uint32_t> v;
   if (pix.empty()) return v; // degenerate case
   size_t base = 0, cur = base + 1;
   for(; cur < pix.size(); ++cur)
      if (pix[cur] != pix[base]) {
         v.push_back(cur - base);
         v.push_back(static_cast<uint32_t>(pix[base]));
         base = cur;
      }
   v.push_back(cur - base);
   v.push_back(static_cast<uint32_t>(pix[base]));
   return v;
}
```

Implementing RLE compression

```
vector<uint32_t> rle_compression(const vector<pixel> &pix) {
   vector<uint32_t> v;
   if (pix.empty()) return v; // degenerate case
   size_t base = 0, cur = base + 1;
   for(; cur < pix.size(); ++cur)
      if (pix[cur] != pix[base]) {
         v.push_back(cur - base);
         v.push_back(static_cast<uint32_t>(pix[base]));
         base = cur;
      }
   v.push_back(cur - base);
   v.push_back(static_cast<uint32_t>(pix[base]));
   return v;
}
```

For this version, the function's signature is quite straightforward (takes a vector<pixel>, returns a vector of integrals). Easy to understand, easy to use

Implementing RLE compression

```
vector<uint32_t> rle_compression(const vector<pixel> &pix) {
   vector<uint32_t> v;
   if (pix.empty()) return v; // degenerate case
   size_t base = 0, cur = base + 1;
   for(; cur < pix.size(); ++cur)
      if (pix[cur] != pix[base]) {
         v.push_back(cur - base);
         v.push_back(static_cast<uint32_t>(pix[base]));
         base = cur;
      }
   v.push_back(cur - base);
   v.push_back(static_cast<uint32_t>(pix[base]));
   return v;
}
```

Comparing with operator!= and converting to a 32 bits unsigned integral seem to be our key operations for this algorithm

Implementing RLE compression

```
vector<uint32_t> rle_compression(const vector<pixel> &pix) {
   vector<uint32_t> v;
   if (pix.empty()) return v; // degenerate case
   size_t base = 0, cur = base + 1;
   for(; cur < pix.size(); ++cur)
      if (pix[cur] != pix[base]) {
         v.push_back(cur - base);
         v.push_back(static_cast<uint32_t>(pix[base]));
         base = cur;
      }
   v.push_back(cur - base);
   v.push_back(static_cast<uint32_t>(pix[base]));
   return v;
}
```

For that reason, we should probably pick a representation for pixel which provides the most efficient operator!= implementation

Implementing RLE compression

```
vector<uint32_t> rle_compression(const vector<pixel> &pix) {
    vector<uint32_t> v;
    if (pix.empty()) return v; // degenerate case
    size_t base = 0, cur = base + 1;
    for(; cur < pix.size(); ++cur)
        if (pix[cur] != pix[base]) {
            v.push_back(cur - base);
            v.push_back(static_cast<uint32_t>(pix[base]));
            base = cur;
        }
    v.push_back(cur - base);
    v.push_back(static_cast<uint32_t>(pix[base]));
    return v;
}
```

The algorithm is simple: pick a reference point (here, the pixel at index base) and look for the first element that is not equivalent to the reference point. This defines a half-open range of equivalent values

Implementing RLE compression

```
vector<uint32_t> rle_compression(const vector<pixel> &pix) {
   vector<uint32_t> v;
   if (pix.empty()) return v; // degenerate case
   size_t base = 0, cur = base + 1;
   for(; cur < pix.size(); ++cur)
      if (pix[cur] != pix[base]) {
         v.push_back(cur - base);
         v.push_back(static_cast<uint32_t>(pix[base]));
         base = cur;
      }
   v.push_back(cur - base);
   v.push_back(static_cast<uint32_t>(pix[base]));
   return v;
}
```

Once that range is identified, we keep the size of that range and the value found in that range (here, converted to an unsigned integral for simplicity)

Implementing RLE compression

```
vector<uint32_t> rle_compression(const vector<pixel> &pix) {
   vector<uint32_t> v;
   if (pix.empty()) return v; // degenerate case
   size_t base = 0, cur = base + 1;
   for(; cur < pix.size(); ++cur)
      if (pix[cur] != pix[base]) {
         v.push_back(cur - base);
         v.push_back(static_cast<uint32_t>(pix[base]));
         base = cur;
      }
   v.push_back(cur - base);
   v.push_back(static_cast<uint32_t>(pix[base]));
   return v;
}
```

There's always a residual range at the end defined by the final sequence or equivalent values. This range is added after the loop and we are done

It works fine, but…

It works fine, but…

- This algorithm works, and probably gives results that tend towards optimality
  - We could do a large number of optimizations, but that’s not the point of this presentation
- However, we cannot use it as is

It works fine, but…

```
vector<uint32_t> rle_compression(const vector<pixel> &pix) {
   vector<uint32_t> v;
   if (pix.empty()) return v; // degenerate case
   size_t base = 0, cur = base + 1;
   for(; cur < pix.size(); ++cur)
      if (pix[cur] != pix[base]) {
         v.push_back(cur - base);
         v.push_back(static_cast<uint32_t>(pix[base]));
         base = cur;
      }
   v.push_back(cur - base);
   v.push_back(static_cast<uint32_t>(pix[base]));
   return v;
}
```

It works fine, but…

Calls to push_back() are amortized constant time, or *O(1) in general*

```
vector<uint32_t> rle_compression(
  vector<uint32_t> v;
  if (pix.empty()) return v; // degen        se
  size_t base = 0, cur = base + 1;
  for(; cur < pix.size(); ++cur)
    if (pix[cur] != pix[base]) {
      v.push_back(cur - base);
      v.push_back(static_cast<uint32_t>(pix[base]));
      base = cur;
    }
  v.push_back(cur - base);
  v.push_back(static_cast<uint32_t>(pix[base]));
  return v;
}
```

It works fine, but…

```
vector<uint32_t> rle_compression(
   vector<uint32_t> v;
   if (pix.empty()) return v; // degen          se
   size_t base = 0, cur = base + 1;
   for(; cur < pix.size(); ++cur)
      if (pix[cur] != pix[base]) {
         v.push_back(cur - base);
         v.push_back(static_cast<uint32_t>(pix[ba
         base = cur;
      }
   v.push_back(cur - base);
   v.push_back(static_cast<uint32_t>(pix[base]));
   return v;
}
```

Calls to push_back() are amortized constant time, or *O(1) in general*

However, *in general* is not sufficient for our needs: v.push_back() could allocate (non-deterministic execution time), and we cannot guarantee the execution of this function will be short enough for our needs

It works fine, but…

```
vector<uint32_t> rle_compression(const vector<pixel> &pix) {
    vector<uint32_t> v;
    if (pix.empty()) return v; // degene
    size_t base = 0, cur = base + 1;
    for(; cur < pix.size(); ++cur)
        if (pix[cur] != pix[base]) {
            v.push_back(cur - base);
            v.push_back(static_cast<uint32_t>(pix[base])
            base = cur;
        }
    v.push_back(cur - base);
    v.push_back(static_cast<uint32_t>(pix[base]));
    return v;
}
```

It’s tempting to reserve() memory right away, but (a) it would remain indeterministic (it’s still an allocation) and (b) we would have to be pessimistic and go for the worst case, which is 2 * pix.size() elements

It works fine, but...

```
vector<uint32_t> rle_compression(const vector<pixel> &pix) {
   vector<uint32_t> v;
   if (pix.empty()) return v; // degenerate case
   size_t base = 0, cur = base + 1;
   for(; cur < pix.size(); ++cur)
      if (pix[cur] != pix[base]) {
         v.push_back(cur - base);
         v.push_back(static_cast<uint32_t>(pix[base
         base = cur;
      }
   v.push_back(cur - base);
   v.push_back(static_cast<uint32_t>(pix[base]));
   return v;
}
```

We also have a linear complexity algorithm, *O(n)* where n==pix.size(). Thus, our running time will (theoretically) be proportional to the number of pixels to check, making it difficult to offer a maximal execution time guarantee

It works fine, but…

- The rle_compression() algorithm is not usable in a context where execution time is strictly delimiter
- To be usable in the context where we want to use it, we will need to modify this algorithm

It works fine, but…

- It has to ensure it will not run for longer than allowed
  - We need a way for the caller to tell us if we can continue

It works fine, but…

- It has to ensure it will not run for longer than allowed
  - We need a way for the caller to tell us if we can continue

A maximum number of iterations?
A deadline?
A maximal execution time?

I favor passing a continuation predicate, which lets the caller in control of whichever way the continuation condition is expressed

It works fine, but…

- It has to ensure it will not run for longer than allowed
  - We need a way for the caller to tell us if we can continue
  - If we want to make progress, we need to know where the most recent call stopped processing, in order to know where to start over on the next call

It works fine, but…

- It has to ensure it will not run for longer than allowed
  - We need a way for the caller to tell us if we can continue
  - If we want to make progress, we need to know where the most recent call stopped processing, in order to know where to start over on the next call

This means we will want to adapt the function's signature somewhat

It works fine, but…

- It has to ensure it will not run for longer than allowed
  - We need a way for the caller to tell us if we can continue
  - If we want to make progress, we need to know where the most recent call stopped processing, in order to know where to start over on the next call
- It has to avoid all underministic execution time behavior
  - In particular, this means « no memory allocation within the function »

It works fine, but…

- It has to ensure it will not run for longer than allowed
  - We need a way for the caller to tell us if we can continue
  - If we want to make progress, we need to know where the most recent call stopped processing, in order to know where to start over on the next call
- It has to avoid all underministic execution time behavior
  - In particular, this means « no memory allocation within the function »

Expressed otherwise, the caller has to supply the output buffer: the callee has no general way to do so *and* respect strict time constraints

RLE compression as a divisible algorithm

```
template <class IIt, class OIt, class Pred>
   std::pair<IIt, OIt> divisible_rle_compression(IIt bs, IIt es, OIt bd, Pred pred) {











   }
```

RLE compression as a divisible algorithm

```
template <class IIt, class OIt, class Pred>
   std::pair<IIt, OIt> divisible_rle_compression(IIt bs, IIt es, OIt bd, Pred pred) {



   }
```

Our new signature is more complex than the original one, which was simply

```
vector<uint32_t> rle_compression(const vector<pixel> &);
```

RLE compression as a divisible algorithm

```
template <class IIt, class OIt, class Pred>
   std::pair<IIt, OIt> divisible_rle_compression(IIt bs, IIt es, OIt bd, Pred pred) {
   }
```

Our input range will be defined by the half-open range [bs,es) (*beginning of the source, end of the source*). For a given sequence to compress, argument bs might change over time as we progress through the input range, wheras es should stay stable

RLE compression as a divisible algorithm

```
template <class IIt, class OIt, class Pred>
   std::pair<IIt, OIt> divisible_rle_compression(IIt bs, IIt es, OIt bd, Pred pred) {

   }
```

We will write to the (current) beginning of the output range, bd (*beginning of destination*). As is customary of output iterators, we will suppose we have been provided a destination large enough to fit our needs, e.g.: an output buffer supplied by the caller, who should be able to determine a reasonable size and allocate – if needed – at an appropriate moment

RLE compression as a divisible algorithm

```
template <class IIt, class OIt, class Pred>
   std::pair<IIt, OIt> divisible_rle_compression(IIt bs, IIt es, OIt bd, Pred pred) {



   }
```

Our continuation predicate pred will be a nullary function (zero arguments) that returns true if we can continue processing. Since we will usually not give guarantees to the caller as to how many times this function will be called, it is typically better if the function is stateless

RLE compression as a divisible algorithm

```
template <class IIt, class OIt, class Pred>
   std::pair<IIt, OIt> divisible_rle_compression(IIt bs, IIt es, OIt bd, Pred pred) {
```

```
// inappropriate (influenced by the number
// of calls made). Gives different results
// if call per compressed sequence or called
// per pixel
[n]() mutable {
   if(n > 0) {
      --n;
      return true;
   }
   return false;
}
```

RLE compression as a divisible algorithm

```
template <class IIt, class OIt, class Pred>
   std::pair<IIt, OIt> divisible_rle_compression(IIt bs, IIt es, OIt bd, Pred pred) {









   }
```

We will return a pair made of where we stopped in the input range and where we stopped in the output range. These values will be fed to the function by the caller in the next call (as arguments bs and bd)

RLE compression as a divisible algorithm

```
template <class IIt, class OIt, class Pred>
  std::pair<IIt, OIt> divisible_rle_compression(IIt bs, IIt es, OIt bd, Pred pred) {
    if (bs == es) return { bs, bd };







  }
```

The trivial case of compressing an empty sequence is covered by returning a pair with bs and bd unchanged

RLE compression as a divisible algorithm

```
template <class IIt, class OIt, class Pred>
  std::pair<IIt, OIt> divisible_rle_compression(IIt bs, IIt es, OIt bd, Pred pred) {
    if (bs == es) return { bs, bd };
    auto base = bs;
    for (++bs; bs != es; ++bs) {
      if (!pred()) return { base, bd };
      if (*base != *bs) {
        *bd++ = distance(base, bs); // how many
        *bd++ = static_cast<std::uint32_t>(*base); // what
        base = bs;
      }
    }
    *bd++ = distance(base, es); // how many
    *bd++ = static_cast<std::uint32_t>(*base); // what
    return { es, bd };
  }
```

The heart of the compression algorithm remains the same: take a reference point, look for a non-equivalent value, then insert how many were found and what the value was in the output range

RLE compression as a divisible algorithm

```
template <class IIt, class OIt, class Pred>
  std::pair<IIt, OIt> divisible_rle_compression(IIt bs, IIt es, OIt bd, Pred pred) {
    if (bs == es) return { bs, bd };
    auto base = bs;
    for (++bs; bs != es; ++bs) {
      if (!pred()) return { base, bd };
      if (*base != *bs) {
        *bd++ = distance(base, bs); // how many
        *bd++ = static_cast<std::uint32_t>(*base); // what
        base = bs;
      }
    }
    *bd++ = distance(base, es); // how many
    *bd++ = static_cast<std::uint32_t>(*base); // what
    return { es, bd };
  }
```

The main difference is that compression can be interrupted at every iteration should the continuation conditions not be met

RLE compression as a divisible algorithm

```
template <class IIt, class OIt, class Pred>
   std::pair<IIt, OIt> divisible_rle_compression(IIt bs, IIt es, OIt bd, Pred pred) {
      if (bs == es) return { bs, bd };
      auto base = bs;
      for (++bs; bs != es; ++bs) {
         if (!pred()) return { base, bd };
         if (*base != *bs) {
            *bd++ = distance(base, bs); // how many
            *bd++ = static_cast<std::uint32_t>(*base); // wha
            base = bs;
         }
      }
      *bd++ = distance(base, es); // how many
      *bd++ = static_cast<std::uint32_t>(*base); // what
      return { es, bd };
   }
```

This choice has consequences. One of them is that we might not make progress over several calls to our function… We might even never make progress if the range to process is long and the available time is short

RLE compression as a divisible algorithm

```
template <class IIt, class OIt, class Pred>
   std::pair<IIt, OIt> divisible_rle_compression(IIt bs, IIt es, OIt bd, Pred pred) {
      if (bs == es) return { bs, bd };
      auto base = bs;
      for (++bs; bs != es; ++bs) {
         if (*base != *bs) {
            *bd++ = distance(base, bs); // how many
            *bd++ = static_cast<std::uint32_t>(*base); // what
            base = bs;
            if (!pred()) return { base, bd };
         }
      }
      *bd++ = distance(base, es); // how many
      *bd++ = static_cast<std::uint32_t>(*base); // what
      return { es, bd };
   }
```

An alternative would be to test the continuation predicate with every subrange instead of every element. This would ensure progress at least once per call, but would risk exceeding the alloted time interval

RLE compression as a divisible algorithm

```
template <class IIt, class OIt, class Pred>
   std::pair<IIt, OIt> divisible_rle_compression(IIt bs, IIt es, OIt bd, Pred pred) {
      if (bs == es) return { bs, bd };
      auto base = bs;
      for (++bs; bs != es; ++bs) {
         if (!pred()) {
            *bd++ = distance(base, bs); // how many
            *bd++ = static_cast<std::uint32_t>(*base); // what
            return { bs, bd };
         }
         if (*base != *bs) {
            // ...
         }
      }
      // ...
      return { es, bd };
   }
```

Another option would be to accept an imperfect compression quality, to let such sequences as *3 R 2 R* instead of *5 R* for example. This would ensure progress on every call too

RLE compression as a divisible algorithm

```
template <class IIt, class OIt, class Pred>
  std::pair<IIt, OIt> divisible_rle_compression(IIt bs, IIt es, OIt bd, Pred pred) {
    if (bs == es) return { bs, bd };
    auto base = bs;
    for (++bs; bs != es; ++bs) {
      if (!pred()) {
        *bd++ = distance(base, bs); // how many
        *bd++ = static_cast<std::uint32_t>(*base); // what
        return { bs, bd };
      }
      if (*base != *bs) {
        // ...
      }
    }
    // ...
    return { es, bd };
  }
```

Another option would be to accept an imperfect compression quality, to let such sequences as *3 R 2 R* instead of *5 R* for example. This would ensure progress on every call too

We could compensate for this suboptimal compression through post-processing of the compressed sequence

RLE compression as a divisible algorithm

```
template <class IIt, class OIt, class Pred>
   std::pair<IIt, OIt> divisible_rle_compression(IIt bs, IIt es, OIt bd, Pred pred) {
      if (bs == es) return { bs, bd };
      auto base = bs;
      for (++bs; bs != es; ++bs) {
         if (!pred()) return { base, bd };
         if (*base != *bs) {
            *bd++ = distance(base, bs); // how many
            *bd++ = static_cast<std::uint32_t>(*base); //
            base = bs;
         }
      }
      *bd++ = distance(base, es); // how many
      *bd++ = static_cast<std::uint32_t>(*base); // what
      return { es, bd };
   }
```

Once again, as was the case with the algorithm in its more traditional form, we need to finish by adding the last sequence of contiguous equivalent values to the compressed sequence. This can be done without testing the continuation predicate: it can be assumed to be a constant-time operation, and can be accounted for in the implementation of pred

RLE compression as a divisible algorithm

```
template <class IIt, class OIt, class Pred>
  std::pair<IIt, OIt> divisible_rle_compression(IIt bs, IIt es, OIt bd, Pred pred) {
    if (bs == es) return { bs, bd };
    auto base = bs;
    for (++bs; bs != es; ++bs) {
      if (!pred()) return { base, bd };
      if (*base != *bs) {
        *bd++ = distance(base, bs); // how many
        *bd++ = static_cast<std::uint32_t>(*base); // what
        base = bs;
      }
    }
    *bd++ = distance(base, es); // how many
    *bd++ = static_cast<std::uint32_t>(*base); // what
    return { es, bd };
  }
```

https://wandbox.org/permlink/zzhVu2stz8giJzIK

Using our divisible algorithm

Using our divisible algorithm

```
int main() {
   vector<pixel> pix;
   for (int i = 0; i != 10; ++i)
      pix.push_back(pixel{ 0, 255, 0 }); // green
   for (int i = 0; i != 4; ++i)
      pix.push_back(pixel{ 0, 0, 255 }); // blue
   for (int i = 0; i != 6; ++i)
      pix.push_back(pixel{ 255, 0, 0 }); // red
   // ...
```

Using our divisible algorithm

To compare how to use the « traditional » and « divisible » forms of our algorithm, let's suppose a small image (ten green pixels, four blue ones and six red ones)

```
int main() {
   vector<pixel> pix;
   for (int i = 0; i != 10; ++i)
      pix.push_back(pixel{ 0, 255, 0 }); // green
   for (int i = 0; i != 4; ++i)
      pix.push_back(pixel{ 0, 0, 255 }); // blue
   for (int i = 0; i != 6; ++i)
      pix.push_back(pixel{ 255, 0, 0 }); // red
   // ...
```

Using our divisible algorithm

Yes, emplace_back() would be better here, but that's not the point ☺

```
int main() {
    vector<pixel> pix;
    for (int i = 0; i != 10; ++i)
        pix.push_back(pixel{ 0, 255, 0 }); // green
    for (int i = 0; i != 4; ++i)
        pix.push_back(pixel{ 0, 0, 255 }); // blue
    for (int i = 0; i != 6; ++i)
        pix.push_back(pixel{ 255, 0, 0 }); // red
    // ...
```

Using our divisible algorithm

```
int main() {
    // ...
    cout << "Before compression:\n\t";
    for (auto p : pix)
        cout << hex << setw(6) << setfill('0')
             << static_cast<uint32_t>(p)
             << dec << ' ';
    cout << endl;
// ...
```

Using our divisible algorithm

```
int main() {
    // ...
    cout << "Before compression:\n\t";
    for (auto p : pix)
        cout << hex << setw(6) << setfill('0')
             << static_cast<uint32_t>(p)
             << dec << ' ';
    cout << endl;
// ...
```

To validate that we have the desired values in our source sequence, we'll examine them displayed in hexadecimal format

Using our divisible algorithm

```
int main() {
    // ...
    cout << "Before compression:\n\t";
    for (auto p
        cout << h
              << s
              <<
    cout << endl
// ...
```

```
Before compression:
      00ff00 00ff00 00ff00 00ff00 00ff00
00ff00 00ff00 00ff00 00ff00 00ff00 ff0000
ff0000 ff0000 ff0000 0000ff 0000ff 0000ff
0000ff 0000ff 0000ff
```

Using our divisible algorithm

```
int main() {
   // ...
   auto v = rle_compression(pix);
   assert(v.size() % 2 == 0);
   cout << "After single call compression:\n\t";
   for (size_t i = 0; i < v.size(); i += 2)
      cout << v[i] << ' '
           << hex << setw(6) << setfill('0')
           << v[i+1] << dec << "\n\t";
   cout << endl;
   // ...
```

Using our divisible algorithm

```
int main() {
   // ...
   auto v = rle_compression(pix);
   assert(v.size() % 2 == 0);
   cout << "After single call compression:\n\t";
   for (size_t i = 0; i < v.size(); i += 2)
      cout << v[i] << ' '
           << hex << setw(6) << setfill('0')
           << v[i+1] << dec << "\n\t";
   cout << endl;
   // ...
```

The « traditional » form is easy to use, as expected

Using our divisible algorithm

```
int main() {
   // ...
   auto v = rle_compression(pix);
   assert(v.size() % 2 == 0);
   cout << "After single call compression:\n\t";
   for (size_t i = 0; i < v.size(); i += 2)
      cout << v[i] << ' '
           << hex << setw(6) << setfill('0')
           << v[i+1] << dec << "\n\t";
   cout << endl;
   // ...
```

The RLE-compressed result necessarily has an even number of values, being made of pairs

Using our divisible algorithm

```
int main() {
   // ...
   auto v = rle_compression(pix);
   assert(v.size() % 2 == 0);
   cout << "After single call compression:\n\t";
   for (size_t i = 0; i < v.size(); i += 2)
      cout << v[i] << ' '
           << hex << setw(6) << setfill('0')
           << v[i+1] << dec << "\n\t";
   cout << endl;
   // ...
```

The results are displayed two-by-two (the size of the subrange is displayed in decimal format, and the pixel value in hexadecimal format)

Using our divisible algorithm

```
int main() {
    // ...
    auto v = rle_compression(p
    assert(v.size() % 2 == 0);
    cout << "After single call compression:\n\t";
    for (size_t i = 0; i < v.size(); i += 2)
        cout << v[i] << ' '
             << hex << setw(6) << setfill('0')
             << v[i+1] << dec << "\n\t";
    cout << endl;
    // ...
```

```
After single call compression:
        10 00ff00
        4 ff0000
        6 0000ff
```

Using our divisible algorithm

```
int main() {
   // ...
   vector<uint32_t> vm;
   auto bs = begin(pix);
   auto bd = back_inserter(vm);
   for (; bs != end(pix); ) {
      auto [bs_, bd_] =
         divisible_rle_compression(bs, end(pix), bd,
                                   [n = 12]() mutable { return --n >= 0; }
         );
      bs = bs_;
      cout << '.' << flush;
   }
// ...
```

Using our divisible algorithm

```
int main() {
   // ...
   vector<uint32_t> vm;
   auto bs = begin(pix);
   auto bd = back_inserter(vm);
   for (; bs != end(pix); ) {
      auto [bs_, bd_] =
         divisible_rle_compression(bs, end(pix), bd,
                                   [n = 12]() mutable { return --n >= 0; }
         );
      bs = bs_;
      cout << '.' << flush;
   }
// ...
```

The divisible version of the algorithm requires that we keep track of our current location in both the source and the destination sequences

Using our divisible algorithm

```
int main() {
   // ...
   vector<uint32_t> vm;
   auto bs = begin(pix);
   auto bd = back_inserter(vm);
   for (; bs != end(pix); ) {
      auto [bs_, bd_] =
         divisible_rle_compression(bs, end(pix), bd,
                                   [n = 12]() mutable { return --n >= 0; }
         );
      bs = bs_;
      cout << '.' << flush;
   }
// ...
```

I used a back_inserter into a vector as destination, but that's only for simplicity: we could have allocated a buffer in the past and kept track of where we are in that buffer instead

Using our divisible algorithm

```
int main() {
   // ...
   vector<uint32_t> vm;
   auto bs = begin(pix);
   auto bd = back_inserter(vm);
   for (; bs != end(pix); ) {
      auto [bs_, bd_] =
         divisible_rle_compression(bs, end(pix), bd,
                                   [n = 12]() mutable { return --n >= 0; }
         );
      bs = bs_;
      cout << '.' << flush;
   }
// ...
```

We progress by small steps. Here, to keep the example simple, I used a number of tests which is not a good way to do this, but it keeps things readable

Using our divisible algorithm

```
int main() {
   // ...
   vector<uint32_t> vm;
   auto bs = begin(pix);
   auto bd = back_inserter(vm);
   for (; bs != end(pix); ) {
      auto [bs_, bd_] =
         divisible_rle_compression(bs, end(pix), bd,
                                   [n = 12]() mutable { return --n >= 0; }
         );
      bs = bs_;
      cout << '.' << flush;
   }
// ...
```

This is just to show that we progress by small steps

Using our divisible algorithm

```
int main() {
    // ...
    assert(vm.size() % 2 == 0);
    cout << "\n\nAfter divisible compression :\n\t";
    for (size_t i = 0; i < vm.size(); i += 2)
        cout << vm[i] << ' '
             << hex << setw(6) << setfill('0')
             << vm[i+1] << dec << "\n\t";
    cout << endl;
}
```

Using our divisible algorithm

```
int main() {
    // ...
    assert(vm.size() % 2 == 0);
    cout << "\n\nAfter divisible compression :\n\t";
    for (size_t i = 0; i < vm.size(); i += 2)
        cout << vm[i] << ' '
             << hex << setw(6) << setfill('0')
             << vm[i+1] << dec << "\n\t";
    cout << endl;
}
```

Here again, the destination sequence has to have an even number of elements

Using our divisible algorithm

```
int main() {
    // ...
    assert(vm.size() % 2 == 0);
    cout << "\n\nAfter divisible compression :\n\t";
    for (size_t i = 0; i < vm.size(); i += 2)
        cout << vm[i] << '
                << hex << setw(
                << vm[i+1] << d
    cout << endl;
}
```

```
..

After divisible compression :
        10 00ff00
        4 ff0000
        6 0000ff
```

Using our divisible algorithm

```
int main() {
    // ...
    assert(vm.size() % 2 == 0);
    cout << "\n\nAfter divisible   pression :\n\t";
    for (size_t i = 0; i < vm. ize(); i += 2)
        cout << vm[i] << '
             << hex << setw(
             << vm[i+1] << d
    cout << endl;
}
```

There are two dots as it took two calls to compress the source sequence

```
..

After divisible compression :
        10 00ff00
        4 ff0000
        6 0000ff
```

Using our divisible algorithm

```
int main() {
    // ...
    assert(vm.size() % 2 == 0);
    cout << "\n\nAfter divisible compression :\n\t";
    for (size_t i = 0; i < vm.size(); i += 2)
        cout << vm[i] << ' '
             << hex << set
             << vm[i+1] <<
    cout << endl;
}
```

https://wandbox.org/permlink/zzhVu2stz8giJzIK

Slowing down to be faster?

Slowing down to be faster?

- The title of this talk is a bit of a (deliberate) misnomer
- We did (indeed!) end up with a slower version of our initial algorithm
  - Slower and more complex!
- That may seem counterintuitive

Slowing down to be faster?

```
// ...
constexpr auto images_second = 60.0;
constexpr auto time_per_image = 1s/images_seco
auto img = prepare_image();
while(!done) {
   const auto pre = high_resolution_clock::now
   display(img);
   auto img = prepare_image();
   const auto post = high_resolution_clock::now();
   const auto remaining = time_per_image - (post - pre);
   this_thread::sleep_for(remaining);
}
// ...
```

What we have done is create a version of our algorithm that can be used within a strict time interval, without using more time than this per execution

Slowing down to be faster?

- From the perspective of the algorithm, we are slower
  - We can be a little slower, or a lot slower
  - We could also never make progress
    - If the sequence we are trying to process takes more time than we have
    - There are sometimes ways around this, but maybe our algorithm is just not the right « fit » for the calling context

Slowing down to be faster?

- From the perspective of the algorithm, we are slower
  - We can be a little slower, or a lot slower
  - We could also never make progress
    - If the sequence we are trying to process takes more time than we have
    - There are sometimes ways around this, but maybe our algorithm is just not the right « fit » for the calling context
- From the perspective of the system, we are more efficient
  - The work we managed to perform instead of sleeping is work we have already done, and will not need to do later

And coroutines?

And coroutines?

- It’s tempting to use coroutines here
  - After all, they are resumable functions
  - They are meant to be interrupted at some point, then resume at a later point where they left off

And coroutines?

- Indeed, they are a good fit
- But… one has to be a bit careful

And coroutines?

```
template <class Pred>
   generator<State> work(State st, Pred pred) {
      for (;;) {
         if(!pred()) co_yield st;
         // difficult task follows (something
         // that requires computing power!)
         this_thread::sleep_for(100ms);
         if(!pred()) co_yield st;
         ++st; // some residual effort
      }
   }
```

And coroutines?

```
template <class Pred>
   generator<State> work(State st,
      for (;;) {
         if(!pred()) co_yield st;
         // difficult task follows (something
         // that requires computing power!)
         this_thread::sleep_for(100ms);
         if(!pred()) co_yield st;
         ++st; // some residual effort
      }
   }
```

Suppose we have a function which performs some critical task…

And coroutines?

```
template <class Pred>
   generator<State> work(State st,
      for (;;) {
         if(!pred()) co_yield st;
         // difficult task follows (something
         // that requires computing power!)
         this_thread::sleep_for(100ms);
         if(!pred()) co_yield st;
         ++st; // some residual effort
      }
   }
```

Suppose we have a function which performs some critical task…

… and might have some time to perform additional work on every iteration

And coroutines?

```
template <class Pred>
   generator<State> work(State st, Pre
      for (;;) {
         if(!pred()) co_yield st;
         // difficult task follows (something
         // that requires computing power!)
         this_thread::sleep_for(100ms);
         if(!pred()) co_yield st;
         ++st; // some residual effort
      }
   }
```

We can test for possible suspension of the ongoing execution at key moments in our iterative cycle. Coroutines make tracking the state of computation trivially easy

And coroutines?

```
template <class Pred>
   generator<State> work(State st, Pred pred) {
      for (;;) {
         if(!pred()) co_yield st;
         // difficult task follows (
         // that requires computing 
         this_thread::sleep_for(100m
         if(!pred()) co_yield st;
         ++st; // some residual effo
      }
   }
```

The problem of this design is that the predicate is passed to the function on the initial call only. Thus, if client code wants to pass a function that checks time constraints, the predicate will have the same state throughout all subsequent calls to the function

And coroutines?

```
int main() {
   auto deadline = high_resolution_clock::now() + 150ms;
   auto pred = [&deadline] {
      return high_resolution_clock::now() < deadline;
   };
   for (auto n : work({}, pred)) {
      if (n >= 10) break;
      cout << n << endl;
      deadline = high_resolution_clock::now() + 150ms;
   }
}
```

And coroutines?

```
int main() {
    auto deadline = high_resolution_clock::now() + 150ms;
    auto pred = [&deadline] {
        return high_resolution_clock::now() < deadline;
    };
    for (auto n : work({}, pred)) {
        if (n >= 10) break;
        cout << n << endl;
        deadline = high_resolution_clock::now() + 150ms;
    }
}
```

A solution to this is to pass the state that has to change over time in the predicate by reference (or by address)

And coroutines?

```
int main() {
    auto deadline = high_resolution_clock::now() + 150ms;
    auto pred = [&deadline] {
        return high_resolution_clock::now() < deadline;
    };
    for (auto n : work({}, pred)) {
        if (n >= 10) break;
        cout << n << endl;
        deadline = high_resolution_clock::now() + 150ms;
    }
}
```

This way, even though the predicate is only actually passed once, in the initial call to the function…

And coroutines?

```
int main() {
    auto deadline = high_resolution_clock::now() + 150ms;
    auto pred = [&deadline] {
        return high_resolution_clock::now() < deadline;
    };
    for (auto n : work({}, pred)) {
        if (n >= 10) break;
        cout << n << endl;
        deadline = high_resolution_clock::now() + 150ms;
    }
}
```

… the caller can update that state in between continuations of the computation

And coroutines?

Of course, this can lead to tricky lifetime management, so make sure the lifetime of the referred-to object within the continuation predicate does not exceed the last call to the coroutine

```
int main() {
    auto deadline = high_resolution_clock::now() + 150ms;
    auto pred = [&deadline] {
        return high_resolution_clock::now() < deadline;
    };
    for (auto n : work({}, pred)) {
        if (n >= 10) break;
        cout << n << endl;
        deadline = high_resolution_clock::now() + 150ms;
    }
}
```

Questions?