Constant Optimizations

Arnar Bjarni Arnarson

Árangursrík forritun og lausn verkefna

School of Computer Science
Reykjavík University

What Are Constant Optimizations?

What Are Constant Optimizations?

- The majority of the course focuses on algorithmic optimizations where time complexity is improved.

What Are Constant Optimizations?

- The majority of the course focuses on algorithmic optimizations where time complexity is improved.
- But time complexity does not tell the whole story since constants are ignored.

What Are Constant Optimizations?

- The majority of the course focuses on algorithmic optimizations where time complexity is improved.
- But time complexity does not tell the whole story since constants are ignored.
- Some complex algorithms have great time complexity but an awful constant factor.

What Are Constant Optimizations?

- The majority of the course focuses on algorithmic optimizations where time complexity is improved.
- But time complexity does not tell the whole story since constants are ignored.
- Some complex algorithms have great time complexity but an awful constant factor.
- Others have “brute force” like complexity but with a good constant factor, less than 1 even.

What Are Constant Optimizations?

- The majority of the course focuses on algorithmic optimizations where time complexity is improved.
- But time complexity does not tell the whole story since constants are ignored.
- Some complex algorithms have great time complexity but an awful constant factor.
- Others have "brute force" like complexity but with a good constant factor, less than 1 even.
- For example, searching for palindromic numbers can be done by iterating through integers in ascending order.

What Are Constant Optimizations?

- The majority of the course focuses on algorithmic optimizations where time complexity is improved.
- But time complexity does not tell the whole story since constants are ignored.
- Some complex algorithms have great time complexity but an awful constant factor.
- Others have "brute force" like complexity but with a good constant factor, less than 1 even.
- For example, searching for palindromic numbers can be done by iterating through integers in ascending order.
- Noticing that all palindromic integers are divisible by 11 allows for a constant optimization, where the time complexity remains the same.

Faster I/O

Faster I/O

Sometimes the slowest part of your program is reading input and writing output. We can limit the impact I/O has on our program by:

Faster I/O

Sometimes the slowest part of your program is reading input and writing output. We can limit the impact I/O has on our program by:

- (C++) using `ios_base::sync_with_stdio(false);` at the start of your program

Faster I/O

Sometimes the slowest part of your program is reading input and writing output. We can limit the impact I/O has on our program by:

- (C++) using `ios_base::sync_with_stdio(false);` at the start of your program
- (C++) reducing flush operations, use `'\n'` instead of `endl`

Faster I/O

Sometimes the slowest part of your program is reading input and writing output. We can limit the impact I/O has on our program by:

- (C++) using `ios_base::sync_with_stdio(false);` at the start of your program
- (C++) reducing flush operations, use `'\n'` instead of `endl`
- (C++) using a custom built function to read integers: LINK

Faster I/O

Sometimes the slowest part of your program is reading input and writing output. We can limit the impact I/O has on our program by:

- (C++) using `ios_base::sync_with_stdio(false);` at the start of your program
- (C++) reducing flush operations, use `'\n'` instead of `endl`
- (C++) using a custom built function to read integers: LINK
- (Python) using `sys.stdin` and `sys.stdout`, and only flushing when needed. Note that `print` and `input` may flush your output arbitrarily, possibly slowing your program unnecessarily.

Faster I/O

Sometimes the slowest part of your program is reading input and writing output. We can limit the impact I/O has on our program by:

- (C++) using `ios_base::sync_with_stdio(false);` at the start of your program
- (C++) reducing flush operations, use `'\n'` instead of `endl`
- (C++) using a custom built function to read integers: LINK
- (Python) using `sys.stdin` and `sys.stdout`, and only flushing when needed. Note that `print` and `input` may flush your output arbitrarily, possibly slowing your program unnecessarily.
- (Java): I do not know how to make it fast, if I/O is a bottleneck, use a different language maybe? Kattio.java exists but is not even comparable to C++ speed.

Locality of Reference

Coming Up!

- Some of the optimizations discussed beyond this point may or may not be done by the compiler depending on the code you write.

Coming Up!

- Some of the optimizations discussed beyond this point may or may not be done by the compiler depending on the code you write.
- The compiler is good at optimizing common patterns, but needs a little guidance sometimes.

Coming Up!

- Some of the optimizations discussed beyond this point may or may not be done by the compiler depending on the code you write.
- The compiler is good at optimizing common patterns, but needs a little guidance sometimes.
- We will be examining a bit how data and instructions are processed by a CPU.

Coming Up!

- Some of the optimizations discussed beyond this point may or may not be done by the compiler depending on the code you write.
- The compiler is good at optimizing common patterns, but needs a little guidance sometimes.
- We will be examining a bit how data and instructions are processed by a CPU.
- Be aware that microbenchmarks such as the ones I made for the slides may not always mean the changes translate perfectly into efficiency for real world code.

Coming Up!

- Some of the optimizations discussed beyond this point may or may not be done by the compiler depending on the code you write.
- The compiler is good at optimizing common patterns, but needs a little guidance sometimes.
- We will be examining a bit how data and instructions are processed by a CPU.
- Be aware that microbenchmarks such as the ones I made for the slides may not always mean the changes translate perfectly into efficiency for real world code.
- The intent is to teach you the ideas. Do not assume that doing things manually guarantees speed increases! Try it out!

Locality of Reference

- The CPU executes instructions, which it needs to load, on data, which it also needs to load.

Locality of Reference

- The CPU executes instructions, which it needs to load, on data, which it also needs to load.
- Modern CPUs tend to separate these two and treat them differently.

Locality of Reference

- The CPU executes instructions, which it needs to load, on data, which it also needs to load.
- Modern CPUs tend to separate these two and treat them differently.
- Recently referenced, or used, data and instructions tend to be reused.

Locality of Reference

- The CPU executes instructions, which it needs to load, on data, which it also needs to load.
- Modern CPUs tend to separate these two and treat them differently.
- Recently referenced, or used, data and instructions tend to be reused.
- Temporal locality: A recently accessed (memory) address is likely to be accessed in the near future.

Locality of Reference

- The CPU executes instructions, which it needs to load, on data, which it also needs to load.
- Modern CPUs tend to separate these two and treat them differently.
- Recently referenced, or used, data and instructions tend to be reused.
- Temporal locality: A recently accessed (memory) address is likely to be accessed in the near future.
- Spatial locality: An address nearby a recently accessed address is likely to be accessed in the near future.

Spotting Locality

```c
int sum(int arr[N]) {
    int result = 0;
    for (int i = 0; i < N; i++) {
        result += a[i];
    }
    return result;
}
```

Spotting Locality

```
int sum(int arr[N]) {
    int result = 0;
    for (int i = 0; i < N; i++) {
        result += a[i];
    }
    return result;
}
```

- Here `result` is referenced in each iteration: temporal locality of data

Spotting Locality

```
int sum(int arr[N]) {
    int result = 0;
    for (int i = 0; i < N; i++) {
        result += a[i];
    }
    return result;
}
```

- Here `result` is referenced in each iteration: temporal locality of data
- Here the same instructions are used in each iteration of a short loop: temporal locality of instructions

Spotting Locality

```c
int sum(int arr[N]) {
    int result = 0;
    for (int i = 0; i < N; i++) {
        result += a[i];
    }
    return result;
}
```

- Here `result` is referenced in each iteration: temporal locality of data
- Here the same instructions are used in each iteration of a short loop: temporal locality of instructions
- Here array elements are accessed in increasing order with no gaps: spatial locality of data

Spotting Locality

```
int sum(int arr[N]) {
    int result = 0;
    for (int i = 0; i < N; i++) {
        result += a[i];
    }
    return result;
}
```

- Here `result` is referenced in each iteration: temporal locality of data
- Here the same instructions are used in each iteration of a short loop: temporal locality of instructions
- Here array elements are accessed in increasing order with no gaps: spatial locality of data
- Here instructions are processed sequentially: spatial locality of instructions

Summing by columns

```
int sum_by_col(int arr[N][M]){
    int result = 0;
    for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

Summing by columns

```
int sum_by_col(int arr[N][M]){
    int result = 0;
    for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- In the inner loop we are jumping between memory addresses in steps of size M.

Summing by columns

```
int sum_by_col(int arr[N][M]){
    int result = 0;
    for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- In the inner loop we are jumping between memory addresses in steps of size M.
- The CPU cache will not be utilized that well, unless it is large enough to store the whole array.

Summing by columns

```
int sum_by_col(int arr[N][M]){
    int result = 0;
    for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- In the inner loop we are jumping between memory addresses in steps of size M.
- The CPU cache will not be utilized that well, unless it is large enough to store the whole array.
- Swapping the order of our loops makes use of spatial locality.

Summing by columns

```
int sum_by_col(int arr[N][M]){
    int result = 0;
    for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- In the inner loop we are jumping between memory addresses in steps of size M.
- The CPU cache will not be utilized that well, unless it is large enough to store the whole array.
- Swapping the order of our loops makes use of spatial locality.
- That way the CPU cache will be utilized better.

Summing by rows

```c
int sum_by_row(int arr[N][M]){
    int result = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- Now the inner loop jumps between addresses in steps of size 1.

Summing by rows

```
int sum_by_row(int arr[N][M]){
    int result = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- Now the inner loop jumps between addresses in steps of size 1.
- I tested with my Intel®Core™i5-4670K

Summing by rows

```
int sum_by_row(int arr[N][M]){
    int result = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- Now the inner loop jumps between addresses in steps of size 1.
- I tested with my Intel®Core™i5-4670K
- Setting $N = M = 10\,000$, so 10^8 additions.

Summing by rows

```
int sum_by_row(int arr[N][M]){
    int result = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- Now the inner loop jumps between addresses in steps of size 1.
- I tested with my Intel®Core™i5-4670K
- Setting $N = M = 10\,000$, so 10^8 additions.
- The function sum_by_col runs in 1.025 seconds.

Summing by rows

```
int sum_by_row(int arr[N][M]){
    int result = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- Now the inner loop jumps between addresses in steps of size 1.
- I tested with my Intel®Core™i5-4670K
- Setting $N = M = 10\,000$, so 10^8 additions.
- The function `sum_by_col` runs in 1.025 seconds.
- The function `sum_by_row` runs in 0.041 seconds.

Summing by rows

```
int sum_by_row(int arr[N][M]){
    int result = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- Now the inner loop jumps between addresses in steps of size 1.
- I tested with my Intel® Core™ i5-4670K
- Setting $N = M = 10\,000$, so 10^8 additions.
- The function `sum_by_col` runs in 1.025 seconds.
- The function `sum_by_row` runs in 0.041 seconds.
- A factor of 25, so not negligible at all! Note that L2 caches generally perform about 25 times faster than RAM.

A Scheduling Problem - From Errichto on CF

Problem description
There are N workers, where $1 \leq N \leq 5\,000$. There is a 30 day window for a two man group project.

Each worker is either available or unavailable each day. You are given the list of days on which each worker is available. The project can only be worked on if both group members are available. You may assume all workers are equally competent, so you only want to maximize the number of days they work together.

What is the best pair of workers to select?

A Scheduling Problem - Slow Solution

```cpp
vector<vector<int>> workers;
int intersection(int a, int b) {
    int i = 0, j = 0;
    int result = 0;
    while (i < N && j < N) {
        if (workers[a][i] == workers[b][j]) {
            result++, i++, j++;
        }
        else if (workers[a][i] < workers[b][j]) {
            i++;
        }
        else {
            j++;
        }
    }
    return result;
}
```

A Scheduling Problem - Slow Solution

```cpp
vector<vector<int>> workers;
int intersection(int a, int b) {
    int i = 0, j = 0;
    int result = 0;
    while (i < N && j < N) {
        if (workers[a][i] == workers[b][j]) {
            result++, i++, j++;
        }
        else if (workers[a][i] < workers[b][j]) {
            i++;
        }
        else {
            j++;
        }
    }
    return result;
}
```

- We can determine the best pair using this function and iterate through all pairs.

A Scheduling Problem - Slow Solution

```cpp
vector<vector<int>> workers;
int intersection(int a, int b) {
    int i = 0, j = 0;
    int result = 0;
    while (i < N && j < N) {
        if (workers[a][i] == workers[b][j]) {
            result++, i++, j++;
        }
        else if (workers[a][i] < workers[b][j]) {
            i++;
        }
        else {
            j++;
        }
    }
    return result;
}
```

- We can determine the best pair using this function and iterate through all pairs.
- The time complexity is $\mathcal{O}(N^2 D)$, in our case $D = 30$.

A Scheduling Problem - Slow Solution

```cpp
vector<vector<int>> workers;
int intersection(int a, int b) {
    int i = 0, j = 0;
    int result = 0;
    while (i < N && j < N) {
        if (workers[a][i] == workers[b][j]) {
            result++, i++, j++;
        }
        else if (workers[a][i] < workers[b][j]) {
            i++;
        }
        else {
            j++;
        }
    }
    return result;
}
```

- We can determine the best pair using this function and iterate through all pairs.
- The time complexity is $\mathcal{O}(N^2D)$, in our case $D = 30$.
- Too slow.

A Scheduling Problem - Less Slow Solution

```cpp
vector<int> workers;
int intersection(int a, int b) {
    int result = 0;
    int inter = workers[a] & workers[b];
    for (int i = 0; i < D; i++) {
        if (inter & 1) {
            result++;
        }
        inter >>= 1;
    }
    return result;
}
```

- Lets store the availability as bitmasks.

A Scheduling Problem - Less Slow Solution

```
vector<int> workers;
int intersection(int a, int b) {
    int result = 0;
    int inter = workers[a] & workers[b];
    for (int i = 0; i < D; i++) {
        if (inter & 1) {
            result++;
        }
        inter >>= 1;
    }
    return result;
}
```

- Lets store the availability as bitmasks.
- This allows us to pack our data more efficiently, which will improve our cache usage.

A Scheduling Problem - Less Slow Solution

```
vector<int> workers;
int intersection(int a, int b) {
    int result = 0;
    int inter = workers[a] & workers[b];
    for (int i = 0; i < D; i++) {
        if (inter & 1) {
            result++;
        }
        inter >>= 1;
    }
    return result;
}
```

- Lets store the availability as bitmasks.
- This allows us to pack our data more efficiently, which will improve our cache usage.
- Probably still too slow.

A Scheduling Problem - Less Slow Solution

```
vector<int> workers;
int intersection(int a, int b) {
    int result = 0;
    int inter = workers[a] & workers[b];
    for (int i = 0; i < D; i++) {
        if (inter & 1) {
            result++;
        }
        inter >>= 1;
    }
    return result;
}
```

- Lets store the availability as bitmasks.
- This allows us to pack our data more efficiently, which will improve our cache usage.
- Probably still too slow.
- The loop still takes $D = 30$ steps, but we can do it faster.

A Scheduling Problem - Magic Solution

```
vector<int> workers;
int intersection(int a, int b) {
    return __builtin_popcount(workers[a] & workers[b]);
}
```

- What sorcery is this?

A Scheduling Problem - Magic Solution

```
vector<int> workers;
int intersection(int a, int b) {
    return __builtin_popcount(workers[a] & workers[b]);
}
```

- What sorcery is this?
- Popcount is the number of set bits, or ones, in the binary representation of the number.

A Scheduling Problem - Magic Solution

```
vector<int> workers;
int intersection(int a, int b) {
    return __builtin_popcount(workers[a] & workers[b]);
}
```

- What sorcery is this?
- Popcount is the number of set bits, or ones, in the binary representation of the number.
- It's time complexity is $\mathcal{O}(1)$ and most CPUs have it as a single instruction.

A Scheduling Problem - Magic Solution

```
vector<int> workers;
int intersection(int a, int b) {
    return __builtin_popcount(workers[a] & workers[b]);
}
```

- What sorcery is this?
- Popcount is the number of set bits, or ones, in the binary representation of the number.
- It's time complexity is $\mathcal{O}(1)$ and most CPUs have it as a single instruction.
- Now our time complexity is $\mathcal{O}(N^2)$ and the code runs fast enough.

A Scheduling Problem - Magic Solution

```
vector<int> workers;
int intersection(int a, int b) {
    return __builtin_popcount(workers[a] & workers[b]);
}
```

- What sorcery is this?
- Popcount is the number of set bits, or ones, in the binary representation of the number.
- It's time complexity is $\mathcal{O}(1)$ and most CPUs have it as a single instruction.
- Now our time complexity is $\mathcal{O}(N^2)$ and the code runs fast enough.
- But wait, there is more!

More Magic

- We have `__builtin_ctz`, which counts trailing zeros.

More Magic

- We have `__builtin_ctz`, which counts trailing zeros.
- We have `__builtin_clz`, which counts leading zeros.

More Magic

- We have `__builtin_ctz`, which counts trailing zeros.
- We have `__builtin_clz`, which counts leading zeros.
- We have `__builtin_parity`, which returns 1 if odd number of bits are set, but 0 if even.

More Magic

- We have `__builtin_ctz`, which counts trailing zeros.
- We have `__builtin_clz`, which counts leading zeros.
- We have `__builtin_parity`, which returns 1 if odd number of bits are set, but 0 if even.
- We have `__builtin_ffs`, which finds the index of the first set bit.

More Magic

- We have `__builtin_ctz`, which counts trailing zeros.
- We have `__builtin_clz`, which counts leading zeros.
- We have `__builtin_parity`, which returns 1 if odd number of bits are set, but 0 if even.
- We have `__builtin_ffs`, which finds the index of the first set bit.
- For `long long` add the suffix `ll`.

More Magic

- We have `__builtin_ctz`, which counts trailing zeros.
- We have `__builtin_clz`, which counts leading zeros.
- We have `__builtin_parity`, which returns 1 if odd number of bits are set, but 0 if even.
- We have `__builtin_ffs`, which finds the index of the first set bit.
- For `long long` add the suffix `ll`.
- Note, that in `C++20` and up, you should use the `std` versions in the header `<bit>`

More Magic

- We have `__builtin_ctz`, which counts trailing zeros.
- We have `__builtin_clz`, which counts leading zeros.
- We have `__builtin_parity`, which returns 1 if odd number of bits are set, but 0 if even.
- We have `__builtin_ffs`, which finds the index of the first set bit.
- For `long long` add the suffix `ll`.
- Note, that in `C++20` and up, you should use the `std` versions in the header `<bit>`
- What if $D = 60$? We could use 64-bit integers.

More Magic

- We have `__builtin_ctz`, which counts trailing zeros.
- We have `__builtin_clz`, which counts leading zeros.
- We have `__builtin_parity`, which returns 1 if odd number of bits are set, but 0 if even.
- We have `__builtin_ffs`, which finds the index of the first set bit.
- For `long long` add the suffix `ll`.
- Note, that in `C++20` and up, you should use the `std` versions in the header `<bit>`
- What if $D = 60$? We could use 64-bit integers.
- What if $D > 64$?

The Infamous Bitset

The Infamous Bitset

- We have seen that representing a set using an integer and bit operations is more efficient than an array of integers.

The Infamous Bitset

- We have seen that representing a set using an integer and bit operations is more efficient than an array of integers.
- Bitwise instructions are generally faster than arithmetic ones.

The Infamous Bitset

- We have seen that representing a set using an integer and bit operations is more efficient than an array of integers.
- Bitwise instructions are generally faster than arithmetic ones.
- Representing a set of integers as bits is also memory efficient and stored sequentially, meaning it is cache friendly.

The Infamous Bitset

- We have seen that representing a set using an integer and bit operations is more efficient than an array of integers.
- Bitwise instructions are generally faster than arithmetic ones.
- Representing a set of integers as bits is also memory efficient and stored sequentially, meaning it is cache friendly.
- To extend this past size 64 we could use `__int128`, but that only takes us so far.

The Infamous Bitset

- We have seen that representing a set using an integer and bit operations is more efficient than an array of integers.
- Bitwise instructions are generally faster than arithmetic ones.
- Representing a set of integers as bits is also memory efficient and stored sequentially, meaning it is cache friendly.
- To extend this past size 64 we could use `__int128`, but that only takes us so far.
- We must also keep in mind the word size w on our machines. Modern machines usually are 64-bit, so $w = 64$.

The Infamous Bitset

- We have seen that representing a set using an integer and bit operations is more efficient than an array of integers.
- Bitwise instructions are generally faster than arithmetic ones.
- Representing a set of integers as bits is also memory efficient and stored sequentially, meaning it is cache friendly.
- To extend this past size 64 we could use `__int128`, but that only takes us so far.
- We must also keep in mind the word size w on our machines. Modern machines usually are 64-bit, so $w = 64$.
- We could make a size $\lceil \frac{D}{w} \rceil$ array using w-bit integers to store our set.

The Infamous Bitset

- We have seen that representing a set using an integer and bit operations is more efficient than an array of integers.
- Bitwise instructions are generally faster than arithmetic ones.
- Representing a set of integers as bits is also memory efficient and stored sequentially, meaning it is cache friendly.
- To extend this past size 64 we could use `__int128`, but that only takes us so far.
- We must also keep in mind the word size w on our machines. Modern machines usually are 64-bit, so $w = 64$.
- We could make a size $\lceil \frac{D}{w} \rceil$ array using w-bit integers to store our set.
- Implementing this includes a lot of shifts and indexing and it is easy to make a mistake and get it wrong.

It Is Our Lucky Day!

- We have in the standard library a data structure known as `bitset`.

It Is Our Lucky Day!

- We have in the standard library a data structure known as `bitset`.
- It is, in essence, a boolean array.

It Is Our Lucky Day!

- We have in the standard library a data structure known as `bitset`.
- It is, in essence, a boolean array.
- How much memory does `bool arr[1 « 30]` take?

It Is Our Lucky Day!

- We have in the standard library a data structure known as `bitset`.
- It is, in essence, a boolean array.
- How much memory does `bool arr[1 << 30]` take?
- Despite booleans only requiring one bit, the data type `bool` is a byte.

It Is Our Lucky Day!

- We have in the standard library a data structure known as `bitset`.
- It is, in essence, a boolean array.
- How much memory does `bool arr[1 « 30]` take?
- Despite booleans only requiring one bit, the data type `bool` is a byte.
- The array is therefore 1024^3 bytes or 1 Gibibyte.

It Is Our Lucky Day!

- We have in the standard library a data structure known as `bitset`.
- It is, in essence, a boolean array.
- How much memory does `bool arr[1 « 30]` take?
- Despite booleans only requiring one bit, the data type `bool` is a byte.
- The array is therefore 1024^3 bytes or 1 Gibibyte.
- Bitsets use the method discussed previously, packing 8 booleans in each byte.

It Is Our Lucky Day!

- We have in the standard library a data structure known as `bitset`.
- It is, in essence, a boolean array.
- How much memory does `bool arr[1 « 30]` take?
- Despite booleans only requiring one bit, the data type `bool` is a byte.
- The array is therefore 1024^3 bytes or 1 Gibibyte.
- Bitsets use the method discussed previously, packing 8 booleans in each byte.
- A `bitset<1 «30>` is therefore 1024^3 bits or 128 Mebibytes.

A Scheduling Problem - Now With Bitsets

```cpp
constexpr int d{ 365 };
constexpr int max_n{ 5'000 };
bitset<d> workers[max_n];
int intersection(int i, int j) {
    return (workers[i] & workers[j]).count();
}
```

A Scheduling Problem - Now With Bitsets

```
constexpr int d{ 365 };
constexpr int max_n{ 5'000 };
bitset<d> workers[max_n];
int intersection(int i, int j) {
    return (workers[i] & workers[j]).count();
}
```

- We must declare the bitset at compile time since the size is given as a template argument.

A Scheduling Problem - Now With Bitsets

```
constexpr int d{ 365 };
constexpr int max_n{ 5'000 };
bitset<d> workers[max_n];
int intersection(int i, int j) {
    return (workers[i] & workers[j]).count();
}
```

- We must declare the bitset at compile time since the size is given as a template argument.
- The bitwise operations of a bitset of size D have complexity $\mathcal{O}\left(\frac{D}{w}\right)$.

A Scheduling Problem - Now With Bitsets

```
constexpr int d{ 365 };
constexpr int max_n{ 5'000 };
bitset<d> workers[max_n];
int intersection(int i, int j) {
    return (workers[i] & workers[j]).count();
}
```

- We must declare the bitset at compile time since the size is given as a template argument.
- The bitwise operations of a bitset of size D have complexity $\mathcal{O}\left(\frac{D}{w}\right)$.
- Total time complexity of the solution using bitsets is $\mathcal{O}\left(N^2 \frac{D}{w}\right)$.

A Scheduling Problem - Now With Bitsets

```cpp
constexpr int d{ 365 };
constexpr int max_n{ 5'000 };
bitset<d> workers[max_n];
int intersection(int i, int j) {
    return (workers[i] & workers[j]).count();
}
```

- We must declare the bitset at compile time since the size is given as a template argument.
- The bitwise operations of a bitset of size D have complexity $\mathcal{O}\left(\frac{D}{w}\right)$.
- Total time complexity of the solution using bitsets is $\mathcal{O}\left(N^2\frac{D}{w}\right)$.
- On a 64-bit machine this is definitely fast enough.

A Scheduling Problem - Now With Bitsets

```
constexpr int d{ 365 };
constexpr int max_n{ 5'000 };
bitset<d> workers[max_n];
int intersection(int i, int j) {
    return (workers[i] & workers[j]).count();
}
```

- We must declare the bitset at compile time since the size is given as a template argument.
- The bitwise operations of a bitset of size D have complexity $\mathcal{O}\left(\frac{D}{w}\right)$.
- Total time complexity of the solution using bitsets is $\mathcal{O}\left(N^2\frac{D}{w}\right)$.
- On a 64-bit machine this is definitely fast enough.
- Practice problems: Chef and Queries, Odd Topic

Branching

Do Repeat Yourself

- Any time you make code reusable, you may be slowing your program down. Generalized code is slower.

Do Repeat Yourself

- Any time you make code reusable, you may be slowing your program down. Generalized code is slower.
- OOP was (is?) very popular and people tend to overuse inheritance, just because they are used to OOP.

Do Repeat Yourself

- Any time you make code reusable, you may be slowing your program down. Generalized code is slower.
- OOP was (is?) very popular and people tend to overuse inheritance, just because they are used to OOP.
- Inheritance can be a massive slowdown, similar to the factor seen before with cache locality. I would advise you to avoid it when you can.

Do Repeat Yourself

- Any time you make code reusable, you may be slowing your program down. Generalized code is slower.
- OOP was (is?) very popular and people tend to overuse inheritance, just because they are used to OOP.
- Inheritance can be a massive slowdown, similar to the factor seen before with cache locality. I would advise you to avoid it when you can.
- Moving a piece of code to a function means you have to call the function, which adds overhead. In C++, the compiler may inline your function, removing the overhead of a function call.

Do Repeat Yourself

- Any time you make code reusable, you may be slowing your program down. Generalized code is slower.
- OOP was (is?) very popular and people tend to overuse inheritance, just because they are used to OOP.
- Inheritance can be a massive slowdown, similar to the factor seen before with cache locality. I would advise you to avoid it when you can.
- Moving a piece of code to a function means you have to call the function, which adds overhead. In C++, the compiler may inline your function, removing the overhead of a function call.
- This is why iterative code is generally faster than recursive code. Some languages support tail recursion which removes the recursive overhead in specific cases.

Do Repeat Yourself

- Any time you make code reusable, you may be slowing your program down. Generalized code is slower.
- OOP was (is?) very popular and people tend to overuse inheritance, just because they are used to OOP.
- Inheritance can be a massive slowdown, similar to the factor seen before with cache locality. I would advise you to avoid it when you can.
- Moving a piece of code to a function means you have to call the function, which adds overhead. In C++, the compiler may inline your function, removing the overhead of a function call.
- This is why iterative code is generally faster than recursive code. Some languages support tail recursion which removes the recursive overhead in specific cases.
- Even loops may slow your code down. Why?

Branching

- You know what else slows down your code? If statements.

Branching

- You know what else slows down your code? If statements.
- They create a branch, such that either the CPU steps to the next address or jumps to an address somewhere else.

Branching

- You know what else slows down your code? If statements.
- They create a branch, such that either the CPU steps to the next address or jumps to an address somewhere else.
- For good cache usage, the likely option should be the step to the next address.

Branching

- You know what else slows down your code? If statements.
- They create a branch, such that either the CPU steps to the next address or jumps to an address somewhere else.
- For good cache usage, the likely option should be the step to the next address.
- The compiler is good at guessing what is likely, but you may guide it using annotations `[[unlikely]]`. This is known as branch prediction.

Branching

- You know what else slows down your code? If statements.
- They create a branch, such that either the CPU steps to the next address or jumps to an address somewhere else.
- For good cache usage, the likely option should be the step to the next address.
- The compiler is good at guessing what is likely, but you may guide it using annotations `[[unlikely]]`. This is known as branch prediction.
- Loops are the same, either you repeat or break from the loop.

Branching

- You know what else slows down your code? If statements.
- They create a branch, such that either the CPU steps to the next address or jumps to an address somewhere else.
- For good cache usage, the likely option should be the step to the next address.
- The compiler is good at guessing what is likely, but you may guide it using annotations `[[unlikely]]`. This is known as branch prediction.
- Loops are the same, either you repeat or break from the loop.
- A higher number of iterations on average with less branching is often faster.

Ternary Sum

```cpp
constexpr int N{ 100'000'000 };
int arr[N];
int regular_loop(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i++) {
        if (arr[i] % 3 == 0) sm -= arr[i];
        else if (arr[i] % 3 == 2) sm += arr[i];
    }
    return sm;
}
```

- This code adds up all the numbers in an array with a slight modification.

Ternary Sum

```
constexpr int N{ 100'000'000 };
int arr[N];
int regular_loop(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i++) {
        if (arr[i] % 3 == 0) sm -= arr[i];
        else if (arr[i] % 3 == 2) sm += arr[i];
    }
    return sm;
}
```

- This code adds up all the numbers in an array with a slight modification.
- The time complexity is linear, but we have 10^8 elements.

Ternary Sum

```cpp
constexpr int N{ 100'000'000 };
int arr[N];
int regular_loop(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i++) {
        if (arr[i] % 3 == 0) sm -= arr[i];
        else if (arr[i] % 3 == 2) sm += arr[i];
    }
    return sm;
}
```

- This code adds up all the numbers in an array with a slight modification.
- The time complexity is linear, but we have 10^8 elements.
- Running this code on my machine takes 0.363 seconds on average.

Ternary Sum

```
constexpr int N{ 100'000'000 };
int arr[N];
int regular_loop(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i++) {
        if (arr[i] % 3 == 0) sm -= arr[i];
        else if (arr[i] % 3 == 2) sm += arr[i];
    }
    return sm;
}
```

- This code adds up all the numbers in an array with a slight modification.
- The time complexity is linear, but we have 10^8 elements.
- Running this code on my machine takes 0.363 seconds on average.
- We can shave off a little by repeating ourselves.

Ternary Sum - Unrolling

```
constexpr int N{ 100'000'000 };
int arr[N];
int unroll_2_loop(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i+=2) {
        if (arr[i] % 3 == 0) sm -= arr[i];
        else if (arr[i] % 3 == 2) sm += arr[i];
        if (arr[i+1] % 3 == 0) sm -= arr[i+1];
        else if (arr[i+1] % 3 == 2) sm += arr[i+1];
    }
    return sm;
}
```

- By repeating ourselves, we reduce branching just a tiny bit.

Ternary Sum - Unrolling

```
constexpr int N{ 100'000'000 };
int arr[N];
int unroll_2_loop(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i+=2) {
        if (arr[i] % 3 == 0) sm -= arr[i];
        else if (arr[i] % 3 == 2) sm += arr[i];
        if (arr[i+1] % 3 == 0) sm -= arr[i+1];
        else if (arr[i+1] % 3 == 2) sm += arr[i+1];
    }
    return sm;
}
```

- By repeating ourselves, we reduce branching just a tiny bit.
- This is known as unrolling loops and your compiler is likely to do this with the `-Ofast` flag.

Ternary Sum - Unrolling

```
constexpr int N{ 100'000'000 };
int arr[N];
int unroll_2_loop(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i+=2) {
        if (arr[i] % 3 == 0) sm -= arr[i];
        else if (arr[i] % 3 == 2) sm += arr[i];
        if (arr[i+1] % 3 == 0) sm -= arr[i+1];
        else if (arr[i+1] % 3 == 2) sm += arr[i+1];
    }
    return sm;
}
```

- By repeating ourselves, we reduce branching just a tiny bit.
- This is known as unrolling loops and your compiler is likely to do this with the `-Ofast` flag.
- Running this code on my machine takes 0.346 seconds on average.

Ternary Sum - Unrolling

```
constexpr int N{ 100'000'000 };
int arr[N];
int unroll_2_loop(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i+=2) {
        if (arr[i] % 3 == 0) sm -= arr[i];
        else if (arr[i] % 3 == 2) sm += arr[i];
        if (arr[i+1] % 3 == 0) sm -= arr[i+1];
        else if (arr[i+1] % 3 == 2) sm += arr[i+1];
    }
    return sm;
}
```

- By repeating ourselves, we reduce branching just a tiny bit.
- This is known as unrolling loops and your compiler is likely to do this with the `-Ofast` flag.
- Running this code on my machine takes 0.346 seconds on average.
- It is consistently faster, but not by much.

Ternary Sum - Unrolling

```
constexpr int N{ 100'000'000 };
int arr[N];
int unroll_2_loop(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i+=2) {
        if (arr[i] % 3 == 0) sm -= arr[i];
        else if (arr[i] % 3 == 2) sm += arr[i];
        if (arr[i+1] % 3 == 0) sm -= arr[i+1];
        else if (arr[i+1] % 3 == 2) sm += arr[i+1];
    }
    return sm;
}
```

- By repeating ourselves, we reduce branching just a tiny bit.
- This is known as unrolling loops and your compiler is likely to do this with the `-Ofast` flag.
- Running this code on my machine takes 0.346 seconds on average.
- It is consistently faster, but not by much.
- Bumping up to 4 elements per iteration gets us to 0.340 seconds. Again, a tiny improvement.

Ternary Sum - Branchless

```
constexpr int N{ 100'000'000 };
int arr[N];
int branchless(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i++) {
        sm += (arr[i] % 3 - 1) * arr[i];
    }
    return sm;
}
```

- Often we can use multiplication or bitmasks as a replacement for arithmetic within if blocks.

Ternary Sum - Branchless

```
constexpr int N{ 100'000'000 };
int arr[N];
int branchless(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i++) {
        sm += (arr[i] % 3 - 1) * arr[i];
    }
    return sm;
}
```

- Often we can use multiplication or bitmasks as a replacement for arithmetic within if blocks.
- This removes the branching within the loop. The only branch is at the end of iteration.

Ternary Sum - Branchless

```
constexpr int N{ 100'000'000 };
int arr[N];
int branchless(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i++) {
        sm += (arr[i] % 3 - 1) * arr[i];
    }
    return sm;
}
```

- Often we can use multiplication or bitmasks as a replacement for arithmetic within if blocks.
- This removes the branching within the loop. The only branch is at the end of iteration.
- Running this code on my machine takes 0.115 seconds on average.

Ternary Sum - Branchless

```
constexpr int N{ 100'000'000 };
int arr[N];
int branchless(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i++) {
        sm += (arr[i] % 3 - 1) * arr[i];
    }
    return sm;
}
```

- Often we can use multiplication or bitmasks as a replacement for arithmetic within if blocks.
- This removes the branching within the loop. The only branch is at the end of iteration.
- Running this code on my machine takes 0.115 seconds on average.
- Branchless programming is a good way to get fast and reliable execution times.

Ternary Sum - Branchless

```
constexpr int N{ 100'000'000 };
int arr[N];
int branchless(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i++) {
        sm += (arr[i] % 3 - 1) * arr[i];
    }
    return sm;
}
```

- Often we can use multiplication or bitmasks as a replacement for arithmetic within if blocks.
- This removes the branching within the loop. The only branch is at the end of iteration.
- Running this code on my machine takes 0.115 seconds on average.
- Branchless programming is a good way to get fast and reliable execution times.
- When writing cryptographic code, it is also a security measure against timing attacks.

Ternary Sum - Unrolled Branchless

```
constexpr int N{ 100'000'000 };
int arr[N];
int branchless_16(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i+=16) {
        sm += (arr[i] % 3 - 1) * arr[i];
        sm += (arr[i+1] % 3 - 1) * arr[i+1];
        // ...
        sm += (arr[i+15] % 3 - 1) * arr[i+15];
    }
    return sm;
}
```

- Unrolling the branchless version like this gets us down to 0.098 seconds.

Ternary Sum - Unrolled Branchless

```
constexpr int N{ 100'000'000 };
int arr[N];
int branchless_16(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i+=16) {
        sm += (arr[i] % 3 - 1) * arr[i];
        sm += (arr[i+1] % 3 - 1) * arr[i+1];
        // ...
        sm += (arr[i+15] % 3 - 1) * arr[i+15];
    }
    return sm;
}
```

- Unrolling the branchless version like this gets us down to 0.098 seconds.
- We have an almost 4 times speedup from the original version.

Ternary Sum - Unrolled Branchless

```
constexpr int N{ 100'000'000 };
int arr[N];
int branchless_16(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i+=16) {
        sm += (arr[i] % 3 - 1) * arr[i];
        sm += (arr[i+1] % 3 - 1) * arr[i+1];
        // ...
        sm += (arr[i+15] % 3 - 1) * arr[i+15];
    }
    return sm;
}
```

- Unrolling the branchless version like this gets us down to 0.098 seconds.
- We have an almost 4 times speedup from the original version.
- Note that if you are dong simple things, the compiler may produce this version from the original version.

Ternary Sum - Unrolled Branchless

```
constexpr int N{ 100'000'000 };
int arr[N];
int branchless_16(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i+=16) {
        sm += (arr[i] % 3 - 1) * arr[i];
        sm += (arr[i+1] % 3 - 1) * arr[i+1];
        // ...
        sm += (arr[i+15] % 3 - 1) * arr[i+15];
    }
    return sm;
}
```

- Unrolling the branchless version like this gets us down to 0.098 seconds.
- We have an almost 4 times speedup from the original version.
- Note that if you are dong simple things, the compiler may produce this version from the original version.
- Another common way to get branchless code is to use ternary operators.

Ternary Sum - Unrolled Branchless

```
constexpr int N{ 100'000'000 };
int arr[N];
int branchless_16(int arr[N]) {
    int sm = 0;
    for (int i = 0; i < N; i+=16) {
        sm += (arr[i] % 3 - 1) * arr[i];
        sm += (arr[i+1] % 3 - 1) * arr[i+1];
        // ...
        sm += (arr[i+15] % 3 - 1) * arr[i+15];
    }
    return sm;
}
```

- Unrolling the branchless version like this gets us down to 0.098 seconds.
- We have an almost 4 times speedup from the original version.
- Note that if you are dong simple things, the compiler may produce this version from the original version.
- Another common way to get branchless code is to use ternary operators.
- Compute both the values a and b and then perform an assignment c

Single Instruction, Multiple Data (SIMD)

SIMD

- You may know that your CPU has 64-bit registers.

SIMD

- You may know that your CPU has 64-bit registers.
- It executes instructions on the contents of the registers.

SIMD

- You may know that your CPU has 64-bit registers.
- It executes instructions on the contents of the registers.
- Your CPU also has larger 256 or 512-bit registers for vectorized instructions.

SIMD

- You may know that your CPU has 64-bit registers.
- It executes instructions on the contents of the registers.
- Your CPU also has larger 256 or 512-bit registers for vectorized instructions.
- This allows you to perform a single instruction on multiple data, potentially speeding up execution by a factor of $4 - 8$.

SIMD

- You may know that your CPU has 64-bit registers.
- It executes instructions on the contents of the registers.
- Your CPU also has larger 256 or 512-bit registers for vectorized instructions.
- This allows you to perform a single instruction on multiple data, potentially speeding up execution by a factor of $4-8$.
- The compiler may vectorize simple patterns for you, such as a simple sum of elements, but in my limited experience you have to get your hands dirty for guarantees.

SIMD

- You may know that your CPU has 64-bit registers.
- It executes instructions on the contents of the registers.
- Your CPU also has larger 256 or 512-bit registers for vectorized instructions.
- This allows you to perform a single instruction on multiple data, potentially speeding up execution by a factor of $4 - 8$.
- The compiler may vectorize simple patterns for you, such as a simple sum of elements, but in my limited experience you have to get your hands dirty for guarantees.
- See documentation for details on functions.

SIMD

```cpp
#include <x86intrin.h> // includes all SIMD intrinsics
constexpr int N{ 100'000'000 };
int arr[N];
int simd(int arr[N]) {
    __m128i sm = _mm_setzero_si128();
    __m128i one = _mm_set_epi32(1, 1, 1, 1);
    for (size_t i = 0; i < N; i+=4) {
        __m128i nums = _mm_loadu_si128((__m128i*) (arr+i));
        __m128i to_add = _mm_set_epi32(arr[i+3]%3, arr[i+2]%3, arr[i+1]%3, arr[i]%3);
        to_add = _mm_sub_epi32(to_add, one);
        to_add = _mm_mullo_epi32(to_add, nums);
        sm = _mm_add_epi32(sm, to_add);
    }
    int res= 0;
    res += _mm_cvtsi128_si32(sm);
    res += _mm_cvtsi128_si32(_mm_bsrli_si128(sm, 4));
    res += _mm_cvtsi128_si32(_mm_bsrli_si128(sm, 8));
    res += _mm_cvtsi128_si32(_mm_bsrli_si128(sm, 12));
    return res;
}
```

- **The example above does not see the best performance gain possible for SIMD. The modulo operation has no vectorized version.**

SIMD

```
#include <x86intrin.h> // includes all SIMD intrinsics
constexpr int N{ 100'000'000 };
int arr[N];
int simd(int arr[N]) {
    __m128i sm = _mm_setzero_si128();
    __m128i one = _mm_set_epi32(1, 1, 1, 1);
    for (size_t i = 0; i < N; i+=4) {
        __m128i nums = _mm_loadu_si128((__m128i*) (arr+i));
        __m128i to_add = _mm_set_epi32(arr[i+3]%3, arr[i+2]%3, arr[i+1]%3, arr[i]%3);
        to_add = _mm_sub_epi32(to_add, one);
        to_add = _mm_mullo_epi32(to_add, nums);
        sm = _mm_add_epi32(sm, to_add);
    }
    int res= 0;
    res += _mm_cvtsi128_si32(sm);
    res += _mm_cvtsi128_si32(_mm_bsrli_si128(sm, 4));
    res += _mm_cvtsi128_si32(_mm_bsrli_si128(sm, 8));
    res += _mm_cvtsi128_si32(_mm_bsrli_si128(sm, 12));
    return res;
}
```

- **The example above does not see the best performance gain possible for SIMD. The modulo operation has no vectorized version.**
- **Despite that, this version takes 0.097 seconds on average, faster than the unrolled branchless code.**

SIMD

```
#include <x86intrin.h> // includes all SIMD intrinsics
constexpr int N{ 100'000'000 };
int arr[N];
int simd(int arr[N]) {
    __m128i sm = _mm_setzero_si128();
    __m128i one = _mm_set_epi32(1, 1, 1, 1);
    for (size_t i = 0; i < N; i+=4) {
        __m128i nums = _mm_loadu_si128((__m128i*) (arr+i));
        __m128i to_add = _mm_set_epi32(arr[i+3]%3, arr[i+2]%3, arr[i+1]%3, arr[i]%3);
        to_add = _mm_sub_epi32(to_add, one);
        to_add = _mm_mullo_epi32(to_add, nums);
        sm = _mm_add_epi32(sm, to_add);
    }
    int res= 0;
    res += _mm_cvtsi128_si32(sm);
    res += _mm_cvtsi128_si32(_mm_bsrli_si128(sm, 4));
    res += _mm_cvtsi128_si32(_mm_bsrli_si128(sm, 8));
    res += _mm_cvtsi128_si32(_mm_bsrli_si128(sm, 12));
    return res;
}
```

- **The example above does not see the best performance gain possible for SIMD. The modulo operation has no vectorized version.**
- **Despite that, this version takes 0.097 seconds on average, faster than the unrolled branchless code.**
- **You can unroll this from 4 computations to 16 to get down to 0.094 seconds.**

SIMD

```cpp
#include <x86intrin.h> // includes all SIMD intrinsics
constexpr int N{ 100'000'000 };
int arr[N];
int simd(int arr[N]) {
    __m128i sm = _mm_setzero_si128();
    __m128i one = _mm_set_epi32(1, 1, 1, 1);
    for (size_t i = 0; i < N; i+=4) {
        __m128i nums = _mm_loadu_si128((__m128i*) (arr+i));
        __m128i to_add = _mm_set_epi32(arr[i+3]%3, arr[i+2]%3, arr[i+1]%3, arr[i]%3);
        to_add = _mm_sub_epi32(to_add, one);
        to_add = _mm_mullo_epi32(to_add, nums);
        sm = _mm_add_epi32(sm, to_add);
    }
    int res= 0;
    res += _mm_cvtsi128_si32(sm);
    res += _mm_cvtsi128_si32(_mm_bsrli_si128(sm, 4));
    res += _mm_cvtsi128_si32(_mm_bsrli_si128(sm, 8));
    res += _mm_cvtsi128_si32(_mm_bsrli_si128(sm, 12));
    return res;
}
```

- **The example above does not see the best performance gain possible for SIMD. The modulo operation has no vectorized version.**
- **Despite that, this version takes 0.097 seconds on average, faster than the unrolled branchless code.**
- **You can unroll this from 4 computations to 16 to get down to 0.094 seconds.**
- **You could however implement a linear search or something without division/modulo and see massive performance gains.**

Summary

- There are many ways to improve performance of a program by constant factors.

Summary

- There are many ways to improve performance of a program by constant factors.
- Most of them rely on improving cache utilization or limiting the number instructions.

Summary

- There are many ways to improve performance of a program by constant factors.
- Most of them rely on improving cache utilization or limiting the number instructions.
- The compiler may do these things for you, but it also may not, you can check after compiling!

Summary

- There are many ways to improve performance of a program by constant factors.
- Most of them rely on improving cache utilization or limiting the number instructions.
- The compiler may do these things for you, but it also may not, you can check after compiling!
- When you do it yourself it is crucial to test whether it is actually faster.

Summary

- There are many ways to improve performance of a program by constant factors.
- Most of them rely on improving cache utilization or limiting the number instructions.
- The compiler may do these things for you, but it also may not, you can check after compiling!
- When you do it yourself it is crucial to test whether it is actually faster.
- Constant time improvements are sometimes easier than algorithmic improvements, and may prove sufficient.