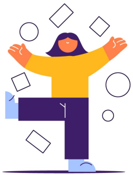
# Applied Algorithms Lec 2: Meet in the Middle

Sam McCauley

September 9, 2025

Williams College

# Admin



- Assignment 0 out!
- Please clone your git repo and fill in the questions
- If you are off-campus, ssh to `lohani.cs.williams.edu`, and then ssh to a lab computer
- I'll post the lab computer addresses once they are set up. Right now, can use Ward lab computers:
  `cow-i23-nuc23.cs.williams.edu`
  `cow-i23-nuc24.cs.williams.edu`
  `...`
  `cow-i23-nuc40.cs.williams.edu`

# Finishing C Review

# Memory Allocation

- `malloc` and `free`
  - Also use `calloc` and `realloc`
  - Need `stdlib.h`
- If you call `C++` code, be careful with mixing `new` and `malloc`
- Use useful library functions like `memset` and `memcpy`
- Example: `memory1.c`

# Variable types

- `int`, `long`, etc. not necessarily the same on different systems
  - On Windows `long` is probably 32 bits, on Mac and Unix it's probably 64 bits
  - `long long` is probably 64 bits

# Variable types

- `int`, `long`, etc. not necessarily the same on different systems
  - On Windows `long` is probably 32 bits, on Mac and Unix it's probably 64 bits
  - `long long` is probably 64 bits

- Instead: include `stdint.h`, describe types explicitly

# Variable types

- `int`, `long`, etc. not necessarily the same on different systems
  - On Windows `long` is probably 32 bits, on Mac and Unix it's probably 64 bits
  - `long long` is probably 64 bits
- Instead: include `stdint.h`, describe types explicitly
- Keep an eye out for unsigned vs signed.

# Variable types

- `int`, `long`, etc. not necessarily the same on different systems
  - On Windows `long` is probably 32 bits, on Mac and Unix it's probably 64 bits
  - `long long` is probably 64 bits
- Instead: include `stdint.h`, describe types explicitly
- Keep an eye out for unsigned vs signed.
- Quick example: `variabletypes.c`

# Variable types

- `int`, `long`, etc. not necessarily the same on different systems
  - On Windows `long` is probably 32 bits, on Mac and Unix it's probably 64 bits
  - `long long` is probably 64 bits
- Instead: include `stdint.h`, describe types explicitly
- Keep an eye out for unsigned vs signed.
- Quick example: `variabletypes.c`
- `printf` does expect primitive types

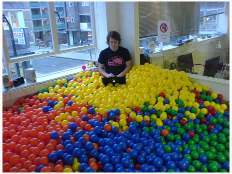
# Variable types cont.

- `int` (etc.) is OK for things like small loops
- If you care *at all* about size you should use the type explicitly
- Up to you when and where you use unsigned
  - Controversial in terms of style
  - Can help with overflow; often changes shift behavior

# List of particularly useful integer variable types

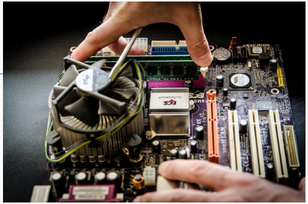
- `int64_t, int32_t`: signed integers of given size
- `uint64_t, uint8_t`: unsigned integers of given size
- `INT64_MAX` (etc.): maximum value of an object of type `int64_t`

# Sorting in C



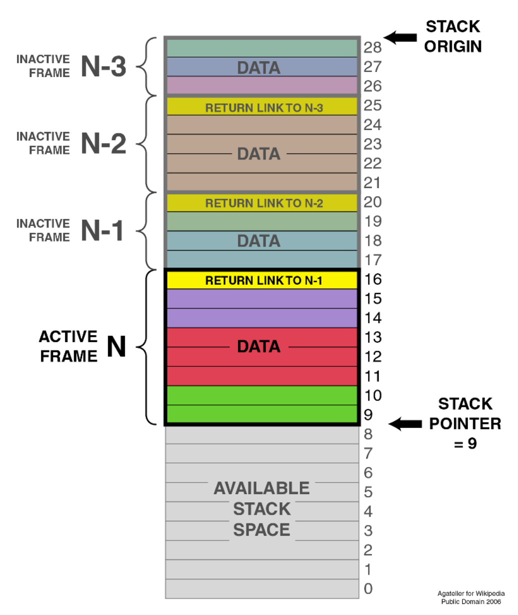
- `qsort()` from `stdlib.h`
- Takes as arguments array pointer, size of array, size of each element, and a comparison function. Let's look at an example of how it works
- What's a downside to this in terms of efficiency?
- Many ways to get better sorts in C:
  - Nicely-written homemade sort
    - Real world: Get an LLM to write one for you. But make sure it works
    - Reminder: *not allowed* in this class; I'll give you a sort for Assignment 1
  - `C++ boost` library
  - Third-party code

# Architecture this Semester



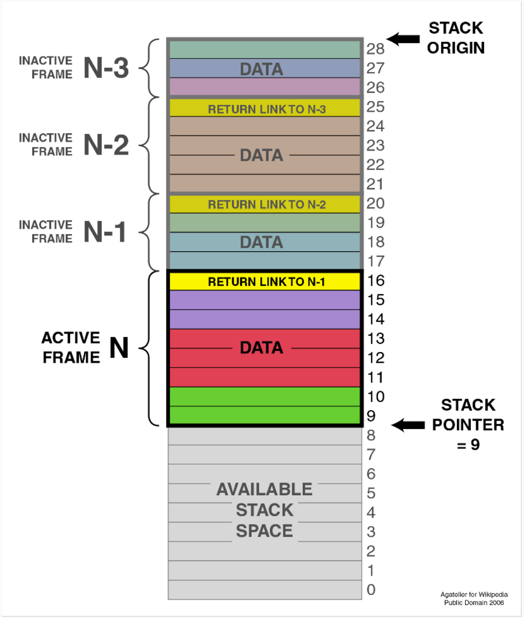
- x86 architecture (not AMD, not M2 etc.)
- Intel i7; run `lscpu` on a lab computer for details
- This *is* likely to have an effect on performance in some cases
- Your home computers are acceptable (but not recommended) for correctness and coarse optimization; use lab computers for fine-grained optimization
- If I ask you to do a performance comparison, you should do it on lab computers. (In the rare case where you don't, you should always write down exactly what machine it was done on.)

# Where are things stored?




- In CPU register (never touching memory)
    - Temporary variables like loop indices
    - Compiler decides this

# Where are things stored?




- In CPU register (never touching memory)
  - Temporary variables like loop indices
  - Compiler decides this
- Call stack
  - Small amount of dedicated memory to keep track of current function and *local* variables
  - Pop back to last function when done
  - **temporary**

# Other place to store things

- The heap!

# Other place to store things

- The heap!
- Very large amount of memory (basically all of RAM)

# Other place to store things

- The heap!
- Very large amount of memory (basically all of RAM)
- Create space on heap using `malloc`

# Other place to store things

- The heap!
- Very large amount of memory (basically all of RAM)
- Create space on heap using `malloc`
- Need `stdlib.h` to use `malloc`

# How to decide stack vs heap?

- Java rules work out well:
    - “objects” and arrays on the heap
    - Anything that needs to be around after the function is over should be on the heap
    - Otherwise declare primitive types and let the compiler work it out
    - Keep scope in mind!

# Makefile

- Each time we change a file, need to recompile that file
- Need to build output file (but don't need to recompile other unchanged files)
- Makefile does this automatically

# In this class

- I'll give you a makefile
- You don't need to change it unless you use multiple files or want to set compiler options
  - Probably don't need to use multiple files in this class
  - (Some exceptions for things like wrapper functions.)

# Let's look quickly at the default Makefile

- `make`, `make clean`, `make debug`

# Compiler flags

- `-g` for debug, `-c` for compile without build (creates `.o` file)

# Compiler flags

- `-g` for debug, `-c` for compile without build (creates `.o` file)
- Different optimization flags:
  - `-O2` is the default level
  - `-O3`, `-Ofast` is more aggressive; doesn't promise correctness in some corner cases
  - `-O0` doesn't optimize; `-Og` is no optimization for debugging
  - Other flags to specifically take advantage of certain compiler features (we'll come back to this)
  - Let's build a (really efficient) Assignment 1 implementation with each flag and see what happens

# Compiler flags

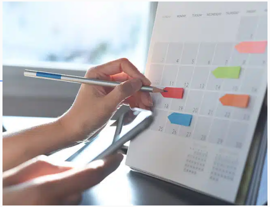
- `-g` for debug, `-c` for compile without build (creates `.o` file)
- Different optimization flags:
  - `-O2` is the default level
  - `-O3`, `-Ofast` is more aggressive; doesn't promise correctness in some corner cases
  - `-O0` doesn't optimize; `-Og` is no optimization for debugging
  - Other flags to specifically take advantage of certain compiler features (we'll come back to this)
  - Let's build a (really efficient) Assignment 1 implementation with each flag and see what happens
- `-S` (along with `-fverbose-asm` for helpful info) to get assembly

# Compiler flags

- `-g` for debug, `-c` for compile without build (creates `.o` file)
- Different optimization flags:
  - `-O2` is the default level
  - `-O3`, `-Ofast` is more aggressive; doesn't promise correctness in some corner cases
  - `-O0` doesn't optimize; `-Og` is no optimization for debugging
  - Other flags to specifically take advantage of certain compiler features (we'll come back to this)
  - Let's build a (really efficient) Assignment 1 implementation with each flag and see what happens
- `-S` (along with `-fverbose-asm` for helpful info) to get assembly
  - Also: "Compiler Explorer" online

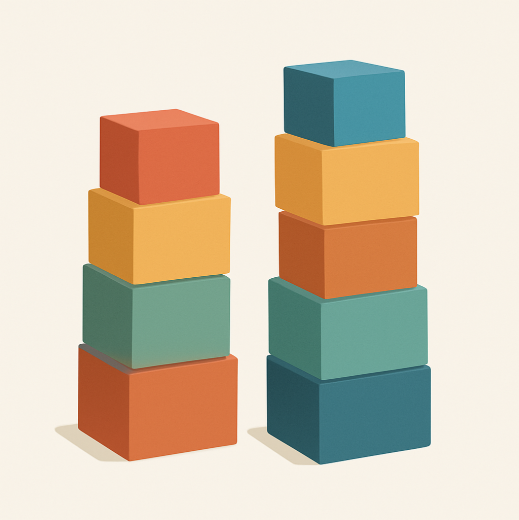
# Meet in the Middle

# Plan for today



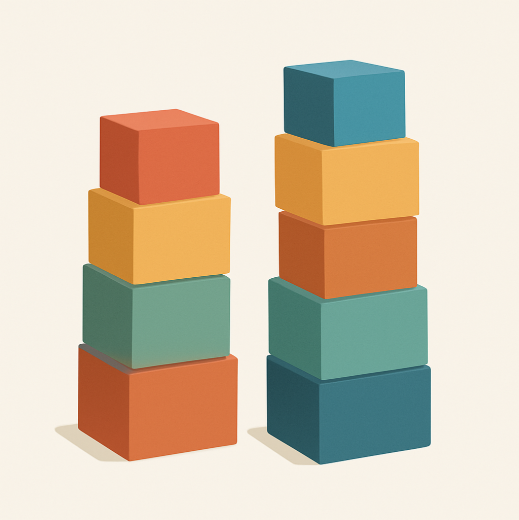
- This part of the course: how time and space interact
- Today: using space to make things run faster
- Specifically, store results of frequently-computed values to save time

# Two towers reminder (?)



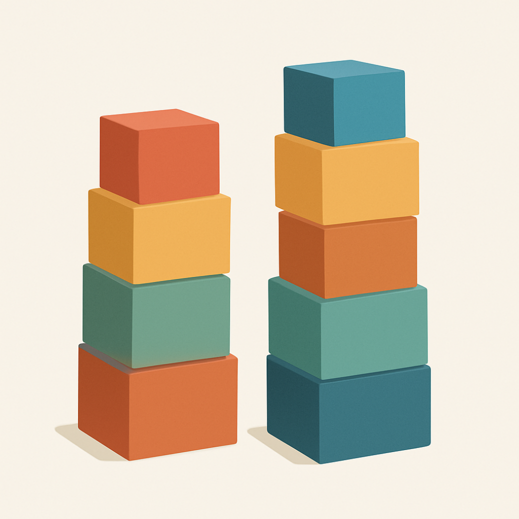
- Input: $n$ square (2D) blocks of given area. Taking the square root of the area gives us the height of each block (let's call the set of heights $S$)

# Two towers reminder (?)



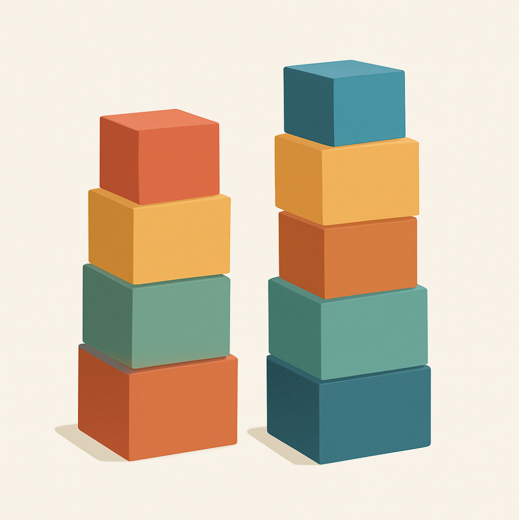
- Input: $n$ square (2D) blocks of given area. Taking the square root of the area gives us the height of each block (let's call the set of heights S)
- (This means the heights are floating-point numbers.)

# Two towers reminder (?)



- Input: $n$ square (2D) blocks of given area. Taking the square root of the area gives us the height of each block (let's call the set of heights S)
- (This means the heights are floating-point numbers.)
- Goal: make two towers with height as close as possible

# Two towers reminder (?)



- Input: $n$ square (2D) blocks of given area. Taking the square root of the area gives us the height of each block (let's call the set of heights $S$)
- (This means the heights are floating-point numbers.)
- Goal: make two towers with height as close as possible
- This is the problem we will solve on Assignment 1

# Let's Rephrase

- Making the towers as *close* as possible is the same as:
- Make the smaller tower as large as possible.

# Let's Rephrase

- Making the towers as *close* as possible is the same as:
- Make the smaller tower as large as possible.
- This means our goal is: find the subset of blocks with largest total height that's at most half the total height; i.e. $\frac{1}{2}\sum_{s\in S} s$.

# Let's Rephrase

- Making the towers as *close* as possible is the same as:
- Make the smaller tower as large as possible.
- This means our goal is: find the subset of blocks with largest total height that's at most half the total height; i.e. $\frac{1}{2}\sum_{s\in S} s$.
- How can we solve this? (Let's go to the board.)

# Let's Rephrase

- Making the towers as *close* as possible is the same as:
- Make the smaller tower as large as possible.
- This means our goal is: find the subset of blocks with largest total height that's at most half the total height; i.e. $\frac{1}{2}\sum_{s \in S} s$.
- How can we solve this? (Let's go to the board.)
  - Try all subsets as the smaller tower; keeping track of largest seen

# Let's Rephrase

- Making the towers as *close* as possible is the same as:
- Make the smaller tower as large as possible.
- This means our goal is: find the subset of blocks with largest total height that's at most half the total height; i.e. $\frac{1}{2}\sum_{s\in S} s$.
- How can we solve this? (Let's go to the board.)
    - Try all subsets as the smaller tower; keeping track of largest seen
- Running time? Space (what do we need to store)?

# Let's Rephrase

- Making the towers as *close* as possible is the same as:
- Make the smaller tower as large as possible.
- This means our goal is: find the subset of blocks with largest total height that's at most half the total height; i.e. $\frac{1}{2}\sum_{s \in S} s$.
- How can we solve this? (Let's go to the board.)
  - Try all subsets as the smaller tower; keeping track of largest seen
- Running time? Space (what do we need to store)?
  - Time: $O(2^n)$. Space: Only need to store current subset; best height seen so far

# Some implementation details

- Can store a subset using an unsigned integer of at most $n$ bits (all instances in Assignment 1 have $n \leq 64$, so `uint64_t` should always work)

# Some implementation details

- Can store a subset using an unsigned integer of at most $n$ bits (all instances in Assignment 1 have $n \leq 64$, so `uint64_t` should always work)

- Then, can iterate through the subsets by starting at `0` and incrementing to $2^n - 1$.

# Some implementation details

- Can store a subset using an unsigned integer of at most $n$ bits (all instances in Assignment 1 have $n \leq 64$, so `uint64_t` should always work)

- Then, can iterate through the subsets by starting at `0` and incrementing to $2^n - 1$.

- For each subset, calculate the height by going through the bits and adding when you see a 1. Keep the heights as an array of floats.

# Meet in the middle

- Divide $S$ into two sets: $S_1$ and $S_2$.

# Meet in the middle

- Divide $S$ into two sets: $S_1$ and $S_2$.
- There must be SOME subset of $S_1$ in the correct final smaller tower.

# Meet in the middle

- Divide $S$ into two sets: $S_1$ and $S_2$.
- There must be SOME subset of $S_1$ in the correct final smaller tower.
- Let's make an algorithm on the board based on this observation. (It won't be faster yet.)

# Meet in the middle

- Divide $S$ into two sets: $S_1$ and $S_2$.

For any set $S'$, let $h(S')$ be the height of all elements in $S'$. Then:

```
for each subset A1 of S1:
    s1 ← h(A1)
    for each subset A2 of S2 :
        if h(A2) + s1 ≤ h(S)/2 :
            updateMax(h(A2) + s1)
```

# Meet in the middle

- Divide $S$ into two sets: $S_1$ and $S_2$.

For any set $S'$, let $h(S')$ be the height of all elements in $S'$. Then:

```
for each subset A_1 of S_1:
    s_1 ← h(A_1)
    for each subset A_2 of S_2 :
        if h(A_2) + s_1 ≤ h(S)/2 :
            updateMax(h(A_2) + s_1)
```

- Running time?

# Meet in the middle

- Divide $S$ into two sets: $S_1$ and $S_2$.

For any set $S'$, let $h(S')$ be the height of all elements in $S'$. Then:

```
for each subset A_1 of S_1:
    s_1 ← h(A_1)
    for each subset A_2 of S_2 :
        if h(A_2) + s_1 ≤ h(S)/2 :
            updateMax(h(A_2) + s_1)
```

- Running time? $2^{n/2} \cdot O(2^{n/2}) = O(2^n)$. Same as before!

# Meet in the Middle

```
for each subset A_1 of S_1:
    s_1 ← h(A_1)
    for each subset A_2 of S_2 :
        if h(A_2) + s_1 ≤ h(S)/2 :
        updateMax(h(A_2) + s_1)
```

# Meet in the Middle

```
for each subset A_1 of S_1:
    s_1 ← h(A_1)
    for each subset A_2 of S_2 :
        if h(A_2) + s_1 ≤ h(S)/2 :
        updateMax(h(A_2) + s_1)
```

- What is this inner poriton doing?

# Meet in the Middle

```
for each subset $A_1$ of $S_1$:
    $s_1 \leftarrow h(A_1)$
    for each subset $A_2$ of $S_2$ :
        if $h(A_2) + s_1 \leq h(S)/2$ :
        updateMax($h(A_2) + s_1$)
```

- What is this inner poriton doing?
- Finds the set $A_2$ with height closest to $h(S)/2$

# Meet in the Middle

```
for each subset A_1 of S_1:
    s_1 ← h(A_1)
    for each subset A_2 of S_2 :
        if h(A_2) + s_1 ≤ h(S)/2 :
            updateMax(h(A_2) + s_1)
```

- What is this inner poriton doing?
- Finds the set $A_2$ with height closest to $h(S)/2$
- How can we preprocess $S_2$ to answer these queries quickly? Let’s split into pairs and discuss for a few minutes.

# Meet in the Middle

```
for each subset A_1 of S_1:
    s_1 ← h(A_1)
    for each subset A_2 of S_2 :
        if h(A_2) + s_1 ≤ h(S)/2 :
        updateMax(h(A_2) + s_1)
```

- What is this inner poriton doing?
- Finds the set $A_2$ with height closest to $h(S)/2$
- How can we preprocess $S_2$ to answer these queries quickly? Let's split into pairs and discuss for a few minutes.
- Answer: we're looking for the largest subset of $S_2$ with height at most $h(S)/2 - s_1$.

# Meet in the Middle

```
for each subset A_1 of S_1:
    s_1 ← h(A_1)
    for each subset A_2 of S_2 :
        if h(A_2) + s_1 ≤ h(S)/2 :
            updateMax(h(A_2) + s_1)
```

- What is this inner poriton doing?
- Finds the set $A_2$ with height closest to $h(S)/2$
- How can we preprocess $S_2$ to answer these queries quickly? Let's split into pairs and discuss for a few minutes.
- Answer: we're looking for the largest subset of $S_2$ with height at most $h(S)/2 - s_1$.
- Sort all subsets of $S_2$. Then can answer this query using binary search!

# Meet in the Middle

```
Fill array P with all subsets of S_2
Sort P by height
for each subset A_1 of S_1:
    s_1 ← h(A_1)
    binsearch(P, h(S)/2 − s_1)
    updateMax(h(A_2) + s_1)
```

- Analysis?

# Meet in the Middle

```
Fill array P with all subsets of S_2
Sort P by height
for each subset A_1 of S_1:
    s_1 ← h(A_1)
    binsearch(P, h(S)/2 − s_1)
    updateMax(h(A_2) + s_1)
```

- Analysis?
- $P$ has length $O(2^{n/2})$. Sorting it takes $O(n2^{n/2})$

# Meet in the Middle

```
Fill array P with all subsets of S_2
Sort P by height
for each subset A_1 of S_1:
    s_1 ← h(A_1)
    binsearch(P, h(S)/2 − s_1)
    updateMax(h(A_2) + s_1)
```

- Analysis?
- $P$ has length $O(2^{n/2})$. Sorting it takes $O(n2^{n/2})$
- Each binary search takes $O(n)$ time; perform $O(2^{n/2})$ of them

# Meet in the Middle

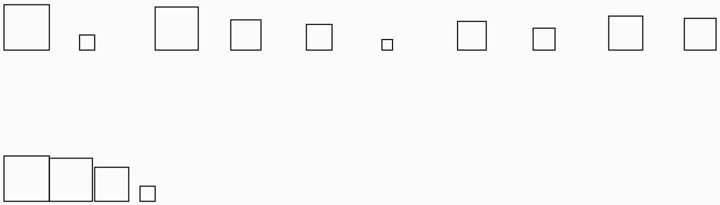
```
Fill array P with all subsets of S_2
Sort P by height
for each subset A_1 of S_1:
    s_1 ← h(A_1)
    binsearch(P, h(S)/2 − s_1)
    updateMax(h(A_2) + s_1)
```

- Analysis?
- $P$ has length $O(2^{n/2})$. Sorting it takes $O(n2^{n/2})$
- Each binary search takes $O(n)$ time; perform $O(2^{n/2})$ of them
- Total: $O(n2^{n/2})$ space, $O(n2^{n/2})$ time

# Meet in the Middle

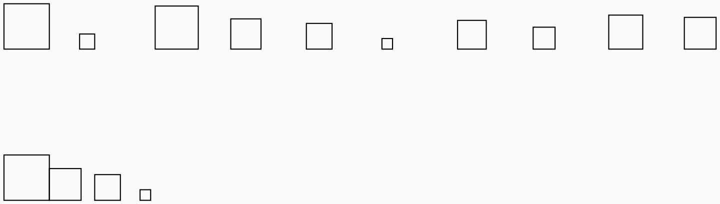
- Before we go forward, let’s go over the high level strategy

# Meet in the Middle



Let’s say we have a set of blocks. Normally we use will try all subsets of these blocks and find the largest that’s at most half the total size.

# Meet in the Middle



Let’s say we have a set of blocks. Normally we use will try all subsets of these blocks and find the largest that’s at most half the total size.

# Meet in the Middle

Partition the blocks into two equal-sized sets.

# Meet in the Middle

Partition the blocks into two equal-sized sets. Question: what subset of the *yellow* blocks is used in the correct solution?

# Meet in the Middle

| | |
|---|---|
| 0.0 | 00000 |
| 7.2 | 00001 |
| 5.1 | 00010 |
| 12.3 | 00011 |
| 9.8 | 00100 |
| 17.0 | 00101 |

...

First, let's do some brute force preprocessing on the blue blocks. Go through all subsets of the blue blocks, and store their heights in a table.

# Meet in the Middle

| | |
|---|---|
| 0.0 | 00000 |
| 5.1 | 00010 |
| 7.2 | 00001 |
| 9.8 | 00100 |

...

First, let's do some brute force preprocessing on the blue blocks. Go through all subsets of the blue blocks, and store their heights in a table. Then, sort the table by height.

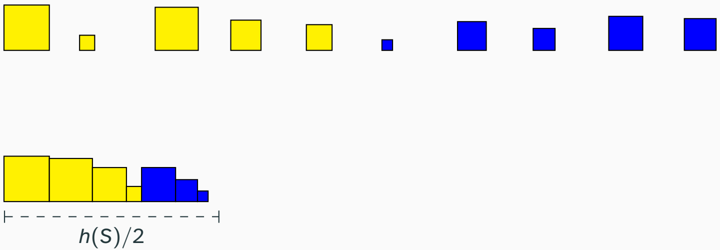
# Meet in the Middle

| | |
|---|---|
| 0.0 | 00000 |
| 5.1 | 00010 |
| 7.2 | 00001 |
| 9.8 | 00100 |

...

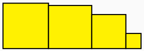
First, preprocess the blue blocks. Go through all subsets of the blue blocks, and store their heights in a table. Then, sort the table by height.

# Meet in the Middle




Now, go through every possible subset of yellow blocks. We want blue blocks with height as close to $h(S)/2 - h(A_1)$ as possible.

# Meet in the Middle



| | |
|---|---|
| 0.0 | 00000 |
| 5.1 | 00010 |
| 7.2 | 00001 |
| 9.8 | 00100 |

...

How quickly can we find the *best* set of blue blocks with height at most $h(S)/2 - h(A_1)$? Why don't we need to check any other subsets of blue blocks?

# What we get

- $O(n2^{n/2})$ space, $O(n2^{n/2})$ time. (Everyone remember how?)

# What we get

- $O(n2^{n/2})$ space, $O(n2^{n/2})$ time. (Everyone remember how?)
- "Meet in the middle"—rather than considering all subsets, we break into two halves. We search in the yellow and blue halves *one at a time*, then combine them to get one solution.

# What we get

- $O(n2^{n/2})$ space, $O(n2^{n/2})$ time. (Everyone remember how?)
- "Meet in the middle"—rather than considering all subsets, we break into two halves. We search in the yellow and blue halves *one at a time*, then combine them to get one solution.
- Very wide uses: optimization problems, cryptography, etc.

# What does this mean?

- What is $O(n2^n)$ vs $O(n2^{n/2})$ time? Do they differ by more than a constant?

# What does this mean?

- What is $O(n2^n)$ vs $O(n2^{n/2})$ time? Do they differ by more than a constant?
- $O(n2^{n/2})$ space is a lot. Is this worth it?

# What does this mean?

- What is $O(n2^n)$ vs $O(n2^{n/2})$ time? Do they differ by more than a constant?
- $O(n2^{n/2})$ space is a lot. Is this worth it?
- Wait, can we do better than this?

# Some questions about meet in the middle

- How else can we store solutions from the blue subproblems? What does this data structure need to support?

# Some questions about meet in the middle

- How else can we store solutions from the blue subproblems? What does this data structure need to support?
    - Needs to support predecessor queries!

# Some questions about meet in the middle

- How else can we store solutions from the blue subproblems? What does this data structure need to support?
  - Needs to support predecessor queries!
- What if we wanted to search for two towers that were exactly equal? Would our strategy change? Could we get improved running time?

# Some questions about meet in the middle

- How else can we store solutions from the blue subproblems? What does this data structure need to support?
  - Needs to support predecessor queries!
- What if we wanted to search for two towers that were exactly equal? Would our strategy change? Could we get improved running time?
- What property must a problem have for MitM to work?
  - Can *all* brute force search problems with $N$ solutions be solved in something like $O(\sqrt{N})$ time?

# Some questions about meet in the middle

- How else can we store solutions from the blue subproblems? What does this data structure need to support?
    - Needs to support predecessor queries!
- What if we wanted to search for two towers that were exactly equal? Would our strategy change? Could we get improved running time?
- What property must a problem have for MitM to work?
    - Can *all* brute force search problems with $N$ solutions be solved in something like $O(\sqrt{N})$ time?
    - No: need the two halves to be *independent*. (We build the table on the blue half once. That table needs to work for every query.)

# Some questions about meet in the middle

- How else can we store solutions from the blue subproblems? What does this data structure need to support?
  - Needs to support predecessor queries!
- What if we wanted to search for two towers that were exactly equal? Would our strategy change? Could we get improved running time?
- What property must a problem have for MitM to work?
  - Can *all* brute force search problems with $N$ solutions be solved in something like $O(\sqrt{N})$ time?
  - No: need the two halves to be *independent*. (We build the table on the blue half once. That table needs to work for every query.)
  - For example, 3SAT doesn't work here. On Assignment 1 you'll look into this further

# Any lingering questions about Assignment 1 or MITM?

# Optimization

# Making Software Fast

What do we mean by fast?

- Generally: runs in a small amount of time (wall-clock time)

# Making Software Fast

What do we mean by fast?

- Generally: runs in a small amount of time (wall-clock time)
- Lots of caveats:

# Making Software Fast

What do we mean by fast?

- Generally: runs in a small amount of time (wall-clock time)
- Lots of caveats:
  - Can be *very* machine-dependent

# Making Software Fast


What do we mean by fast?

- Generally: runs in a small amount of time (wall-clock time)
- Lots of caveats:
  - Can be *very* machine-dependent
  - May depend on the structure of the data

# Making Software Fast


What do we mean by fast?

- Generally: runs in a small amount of time (wall-clock time)
- Lots of caveats:
  - Can be *very* machine-dependent
  - May depend on the structure of the data
  - May depend on random choices of the program

# Making Software Fast


What do we mean by fast?

- Generally: runs in a small amount of time (wall-clock time)
- Lots of caveats:
    - Can be *very* machine-dependent
    - May depend on the structure of the data
    - May depend on random choices of the program
    - Can be impacted by other software running on the computer

# Making Software Fast


What do we mean by fast?

- Generally: runs in a small amount of time (wall-clock time)
- Lots of caveats:
  - Can be *very* machine-dependent
  - May depend on the structure of the data
  - May depend on random choices of the program
  - Can be impacted by other software running on the computer
  - May depend on the compiler used

# Making Software Fast

What do we mean by fast?

- Generally: runs in a small amount of time (wall-clock time)
- Lots of caveats:
  - Can be *very* machine-dependent
  - May depend on the structure of the data
  - May depend on random choices of the program
  - Can be impacted by other software running on the computer
  - May depend on the compiler used
  - Oftentimes: not obvious what causes the improvement!

# What units to measure time?

# What units to measure time?

- Overall: CPU time
  - Some idiosyncracies in how we’re measuring it
  - CPU vs wall clock time shouldn’t make much difference for us
  - Parallelism doesn’t help

# What units to measure time?

- Overall: CPU time
  - Some idiosyncracies in how we're measuring it
  - CPU vs wall clock time shouldn't make much difference for us
  - Parallelism doesn't help

- Costs of specific operations are sometimes given using number of "CPU cycles"

# What units to measure time?

- Overall: CPU time
  - Some idiosyncracies in how we're measuring it
  - CPU vs wall clock time shouldn't make much difference for us
  - Parallelism doesn't help

- Costs of specific operations are sometimes given using number of "CPU cycles"

- Not-really-accurate-anymore definition of a cycle: time to perform one basic operation

# Easiest way to measure time: just time it using built-in tools!

Easy, probably reflective of what you want.

But some things to bear in mind:

- Make sure your timing is macroscopic.
  - No timing is exact.
  - CPU clocks usually only have a resolution of $\approx$ 1 million ticks per second (sometimes less)
  - Minimize issues with overhead, external factors
  - Rule of thumb: ideally an experiment will take $\approx$ 1 second
  - Always repeat several times and check consistency

# Easiest way to measure time: just time it using built-in tools!

Easy, probably reflective of what you want.

But some things to bear in mind:

- Make sure your timing is macroscopic.
  - No timing is exact.
  - CPU clocks usually only have a resolution of $\approx$ 1 million ticks per second (sometimes less)
  - Minimize issues with overhead, external factors
  - Rule of thumb: ideally an experiment will take $\approx$ 1 second
  - Always repeat several times and check consistency

- Let's look at how `test.c` times your code on Homework 1

# Easiest way to measure time: just time it using built-in tools!

Easy, probably reflective of what you want.

But some things to bear in mind:

- Make sure your timing is macroscopic.
  - No timing is exact.
  - CPU clocks usually only have a resolution of $\approx$ 1 million ticks per second (sometimes less)
  - Minimize issues with overhead, external factors
  - Rule of thumb: ideally an experiment will take $\approx$ 1 second
  - Always repeat several times and check consistency
- Let's look at how `test.c` times your code on Homework 1
- Can also use unix `time` function

# Making Software Fast

In this class:

- Measure time using built-in tools

# Making Software Fast

In this class:

- Measure time using built-in tools
- On lab machines

# Making Software Fast

In this class:

- Measure time using built-in tools
- On lab machines
- Always be specific about datasets

# Making Software Fast

In this class:

- Measure time using built-in tools
- On lab machines
- Always be specific about datasets
- Run multiple times to try to avoid issues where other software interferes, or we get lucky/unlucky with random choices

# Making Software Fast

In this class:

- Measure time using built-in tools
- On lab machines
- Always be specific about datasets
- Run multiple times to try to avoid issues where other software interferes, or we get lucky/unlucky with random choices
- Look for *significant* improvement: usually you want experiments where the running time is close to a second, or more

# Making Software Fast

In this class:

- Measure time using built-in tools
- On lab machines
- Always be specific about datasets
- Run multiple times to try to avoid issues where other software interferes, or we get lucky/unlucky with random choices
- Look for *significant* improvement: usually you want experiments where the running time is close to a second, or more
    - Very difficult to draw conclusions from “Program A ran in 10 milliseconds; Program B ran in 15 milliseconds.”

**Let’s say I have a piece of code, and I want to make it faster. How should I do that?**

# Thought question

- What part of a program is most important to speed up?

# Thought question

- What part of a program is most important to speed up?
- Let’s say I have several functions. How can I choose which to try to optimize first?

# Thought question

- What part of a program is most important to speed up?
- Let's say I have several functions. How can I choose which to try to optimize first?
- Answer: the one that takes the most *total* time
  - Time it takes $\times$ number of times it's called
  - May not be the slowest function—in fact, it's often a very fast but very frequently-used function

# Thought question

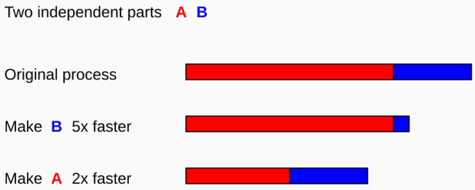
- What part of a program is most important to speed up?

- Let's say I have several functions. How can I choose which to try to optimize first?

- Answer: the one that takes the most *total* time
  - Time it takes $\times$ number of times it's called
  - May not be the slowest function—in fact, it's often a very fast but very frequently-used function

- Probably need to take into account potential to speed it up as well—I want the function that takes up the most time that I can save.

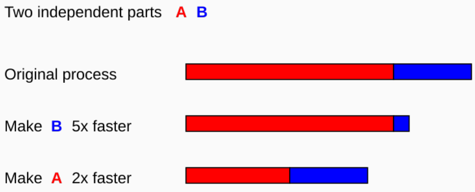
# Amdahl's Law




If a function takes up a $p$ fraction of the entire program's runtime, and you speed it up by a factor $s$, then the overall program speeds up by a factor

$$\frac{1}{1-p+p/s}$$

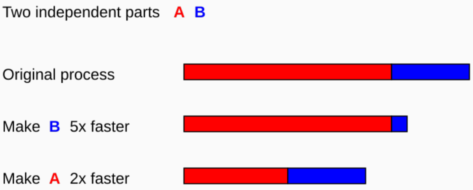
# Amdahl's Law




If a function takes up a $p$ fraction of the entire program's runtime, and you speed it up by a factor $s$, then the overall program speeds up by a factor

$$\frac{1}{1-p+p/s}$$

# Amdahl's Law




If a function takes up a fraction of the entire program's runtime and you speed it up by a factor s, then the overall program speeds up by a factor

$$\frac{1}{1-p+p/s}$$

I want you to know the principle here, but not memorize the formula.

# Amdahl's Law and Asymptotics

- Can *estimate* the total time of an algorithm asymptotically

# Amdahl's Law and Asymptotics

- Can *estimate* the total time of an algorithm asymptotically
- Example: Where to improve Dijkstra's algorithm?

# Dijkstra's Algorithm

```
function Dijkstra(Graph, source):
    create vertex set Q
    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0
    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q
        for each neighbor v of u still in Q:
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u
    return dist[], prev[]
```

# Dijkstra’s Algorithm

```
function Dijkstra(Graph, source):
    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q
        for each neighbor v of u still in Q:
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u
    return dist[], prev[]
```

The inner for loop (blue part) is, at first glance, by far the most important part to optimize.

# Timing one portion of your code

For Amdahl’s, we want to time the total time a subroutine takes over all calls. How can we hope to do that if each call is very fast?

# Timing one portion of your code

For Amdahl’s, we want to time the total time a subroutine takes over all calls. How can we hope to do that if each call is very fast?

1. One option: factor out subroutine using separate testing code
    - Need to get info on how often it’s called; simulate correct types of data.
    - Make sure the compiler does not optimize out your whole experiment!

# Timing one portion of your code

For Amdahl's, we want to time the total time a subroutine takes over all calls. How can we hope to do that if each call is very fast?

1. One option: factor out subroutine using separate testing code
    - Need to get info on how often it's called; simulate correct types of data.
    - Make sure the compiler does not optimize out your whole experiment!
2. Another option: Run same code with and without subroutine
    - Does that change the data the function is called with? Will the change in data affect running time?

# Timing one portion of your code

For Amdahl's, we want to time the total time a subroutine takes over all calls. How can we hope to do that if each call is very fast?

1. One option: factor out subroutine using separate testing code
   - Need to get info on how often it's called; simulate correct types of data.
   - Make sure the compiler does not optimize out your whole experiment!
2. Another option: Run same code with and without subroutine
   - Does that change the data the function is called with? Will the change in data affect running time?
3. Profiling! Tools that will time your functions for you.

We'll come back to this with some examples later this week. Bear in mind: benchmarking itself is an entire area of computer science.

# Optimization

## Code Profiling

# Profiling code

- Why not just have your computer tell you what functions are caused the most, or keep track of how long they run, or monitor specific high-cost operations?
- Lots of such tools! We'll look at a couple of them right now, and use them throughout the class.
  - `gprof`
  - `cachegrind`
  - We won't use `perf` but some people like it
  - We won't use Intel VTune either but seems very cool and powerful
- What do you think some advantages and disadvantages are of using profiling software?

# gprof

- Older command line tool

# gprof

- Older command line tool
- Uses sampling to collect data

# `gprof`

- Older command line tool
- Uses sampling to collect data
- Designed to talk with `gcc` using `-pg` flag

# `gprof`

- Older command line tool
- Uses sampling to collect data
- Designed to talk with `gcc` using `-pg` flag
- Gives information about time as well as the call graph

# `gprof`

- Older command line tool
- Uses sampling to collect data
- Designed to talk with `gcc` using `-pg` flag
- Gives information about time as well as the call graph
- *Quite* limited. But in some circumstances gives good advice.
    - Recursion; function-level resolution; cannot optimize; overhead; sampling problems

# `gprof`

- Older command line tool
- Uses sampling to collect data
- Designed to talk with `gcc` using `-pg` flag
- Gives information about time as well as the call graph
- *Quite* limited. But in some circumstances gives good advice.
  - Recursion; function-level resolution; cannot optimize; overhead; sampling problems
- We’ll look at some examples later

# `callgrind` and `cachegrind`

- Features of `valgrind`

# `callgrind` and `cachegrind`

- Features of `valgrind`
- `callgrind` gives `gprof`-like profiling

# `callgrind` and `cachegrind`

- Features of `valgrind`
- `callgrind` gives `gprof`-like profiling
- `cachegrind` helps determine the cost of *moving* data: cache misses, branch mispredictions, etc.

# `callgrind` and `cachegrind`

- Features of `valgrind`
- `callgrind` gives `gprof`-like profiling
- `cachegrind` helps determine the cost of *moving* data: cache misses, branch mispredictions, etc.
- Essentially runs the program on a virtual machine

# `callgrind` and `cachegrind`

- Features of `valgrind`
- `callgrind` gives `gprof`-like profiling
- `cachegrind` helps determine the cost of *moving* data: cache misses, branch mispredictions, etc.
- Essentially runs the program on a virtual machine
- Gives information about costs you could not otherwise get, but VERY slow.

# Amdahl's Law Takeaways

- When optimizing code, always think first about where your effort is best spent!
- Looking for the portion of the code with most total time
- Can estimate using asymptotics. And/or, run experiments.

# Thought Question in Pairs

We want to make our code faster. What makes code fast? What makes a specific function fast? What slows it down?

# Things to consider

- Good asymptotics (avoid doing too many operations)

# Things to consider

- Good asymptotics (avoid doing too many operations)
- Expensive vs cheap operations

# Things to consider

- Good asymptotics (avoid doing too many operations)
- Expensive vs cheap operations
- Branch mispredictions

# Things to consider

- Good asymptotics (avoid doing too many operations)
- Expensive vs cheap operations
- Branch mispredictions
- Next topic: *cache efficiency*