Applied Algorithms Lec 2: Meet in the Middle and Optimization

Sam McCauley

September 10, 2024

Williams College

Admin

- Office hours 2-5 tomorrow and 3-5 Thursday in TCL 306
- Do Assignment 0 if you haven't
- Assignment 1 released tomorrow; we'll talk about the assignment and handin instructions on Friday
- Comment about Assignments: they can have (brief) questions about techniques seen on homework, but the code and main focus is on a new, related algorithm

Plan going forward

- Today: wrap up C review; start first part of course
    - Part 1: Time and space
    - Today is “meet in the middle”—topic of Assignment 1
- Friday: optimization, handing in assignments
- After that: how to analyze cache misses *algorithmically*

Wrapping up 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`

Sorting in C

- `qsort()` from `stdlib.h`

- Takes as arguments array pointer, size of array, size of each element, and a comparison function

- What's a downside to this in terms of efficiency?

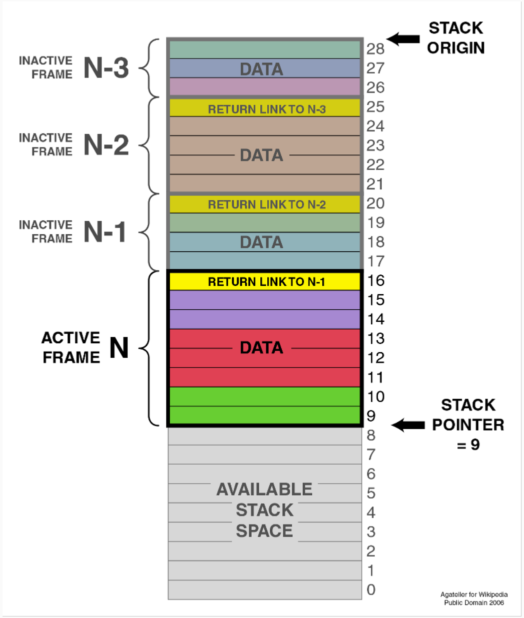
- Many ways to get better sorts in C:
    - Nicely-written homemade sort
    - `C++ boost` library
    - Third-party code

- Instructions to get this to work in handouts on the website (**strictly optional**)

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 fine for correctness and coarse optimization; use lab computers for fine-grained optimization
- If I ask you to do a performance comparison, you should generally do it on lab computers. In any case you should write what you do it on.

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!
- 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)
- 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)
- `-S` (along with `-fverbose-asm` for helpful info) to get assembly
- Also: "Compiler Explorer" online

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

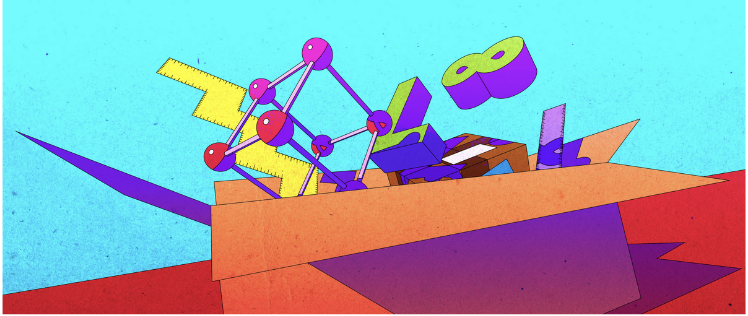
Variable types cont.

- `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

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`

Part 1: Time and Space



How Space and Time Shape Algorithms

By BEN BRUBAKER

Do you ever strategize about ways to speed up your commute, repack a suitcase to free up some room, or tweak a recipe to get food on the table faster? That's

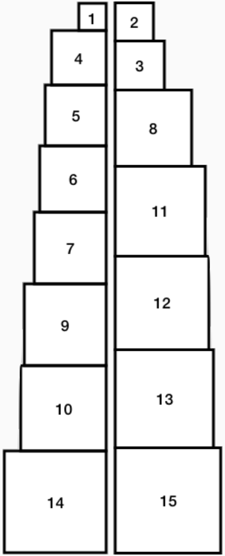
Article in Quanta magazine yesterday about time and space in algorithms.

Part 1 of the course

- In CS 256, we focused largely on the running time of an algorithm, and occasionally talked about space
- But: the way time and space interact is crucial to understanding algorithmic efficiency
- Next three weeks: explore time and space in more detail, using a couple classic algorithms as examples of:
    1. How using more space can decrease running time bounds;
    2. How using *higher* running time bounds to improve space efficiency can decrease wall-clock running time;
    3. How time and space trade off with cache efficiency

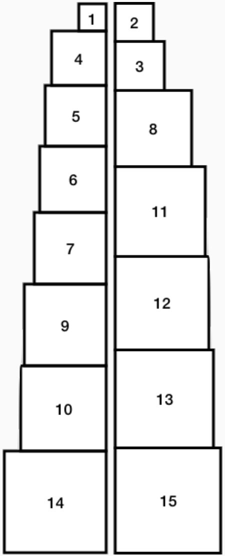
Meet in the Middle

Two towers reminder (?)



- Input: n 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)
- Goal: make two towers with height as close as possible

Two towers observations from 136



- Equivalent problem: 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 $\frac{1}{2}\sum_{s \in S} s$.
- Any ideas for how to solve this correctly (but slowly)?
- First method: try all subsets. For each, calculate its height; store best seen at each point.
- Running time? Space?
- $O(n2^n)$ time; $O(n)$ space

Some implementation details

- Can store a subset using an int of at most n bits (all instances have $n \leq 64$)
    - Each `0` means the item is not in the set; each `1` means the item is in the set
    - Let's do a quick example on the board

- 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.

- Then only need $O(1)$ space (just store 1 integer at all times)

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.
- On board: how can we use this to design a algorithm? (Not fast yet!)

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

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

- What is the inner loop doing?
- Finds the set A_2 with height closest to $h(S)/2$
- How can we preprocess S_2 to answer these queries quickly?
- Answer: sort all subsets of S_2. Then can answer this query using binary search!

Meet in the Middle

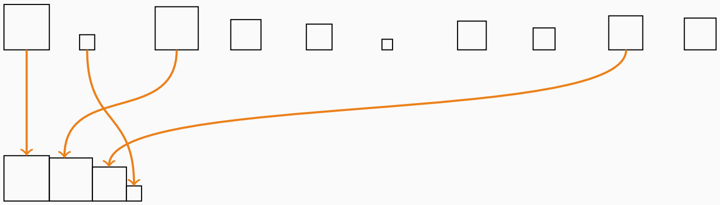
```
Fill array P with all subsets of S_2
Sort P by height
for each subset A_1 of S_1:
    s_1 ← h(A_1)
    binsearch(P, h(S)/2 − s_1)
    updateMax(h(A_2) + s_1)
```

- Let's analyze this approach.
- 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 subset that's at most half the total size.

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
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.
Then, sort the table by height.

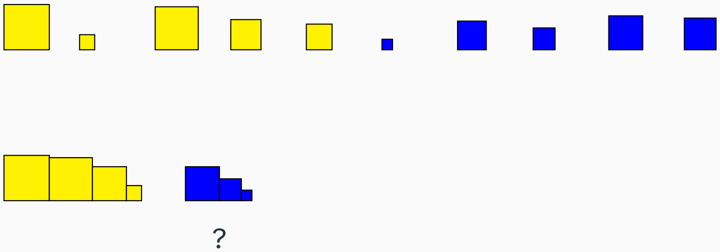
Meet in the Middle

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

...

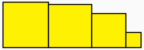
First, let's do some brute force preprocessing on the blue blocks.
Go through all subsets of the blue blocks, and store their heights in a table.
Then, sort the table by height.

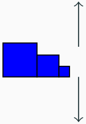
Meet in the Middle



Now, go through every possible set of yellow blocks.
If the yellow blocks have height $h(A_1)$, 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

...

Now, go through every possible set of yellow blocks. How quickly can we find the *best* set of blue blocks? Why don't we need to check any other subsets of blue blocks?

What we get

- $O(2^{n/2})$ 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?
- $O(2^{n/2})$ space is a lot. Is this worth it?
- Wait, can we do better than this?

Some questions about meet in the middle

- How can we store the 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 you'll see another problem where there are issues.

Optimization thought questions

- The data we're sorting has a special structure. Can we use that structure to improve the sort?
- Figuring out the size of a tower is expensive. Can we make this cost less than $O(n)$? Do these changes have other costs?
- Binary search has many branch mispredictions and is cache-inefficient. (We'll talk about these terms more next lecture.) Is there a way to solve the problem without binary search, improving cache efficiency? Or to avoid some of these costs with the binary search?

Meet in the Middle

- A way to use extra space to dramatically improve the *running time* of some search algorithms

- Any lingering questions about meet in the middle?

- Assignment 1 released tonight; I'll set up starter repos by tomorrow evening (fill out the Assignment 0 form if you haven't!)

Principles of Optimization

Reminder

- “Premature optimization is the root of all evil!”
- Don’t optimize your code until you have a working copy.
- Some gray area with structural decisions/trivial ideas—but until something works that is your main goal.

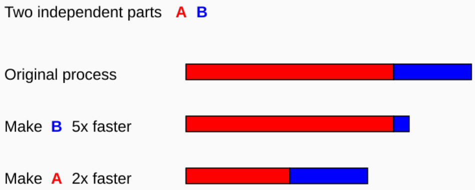
Theory and Reality

- Computers are complicated! (And processors are proprietary!)
- Efficiency is always going to be highly experimental.
  - Sometimes something should work, but doesn't. Or vice versa.
- Goal for this section: *better* understanding of where costs come from and how we can measure them

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

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

- Examples