Applied Algorithms Lec 3: Meet in the Middle

Sam McCauley

October 21, 2021

Williams College

Admin

- Assignment 1 out!
- Github access granted. Testing starts tonight.
- Let me know as soon as possible if there are any issues
- Start early! (Especially on the questions.)
- Next Tuesday's office hours moved to Wednesday 2-4
  - Let me know if you can't make it.
- Don't turn off the machines in TCL 312 (Ctl alt delete logs out)

Books are available in lab

- Here they are!

Submitting Assignments

Basics of submission

- Each of you have a git repo
- Your code is automatically run twice a day on TCL312 machines
- Feedback from automatic runs given back to you in `feedback/` directory
- Also some thought questions in `handout/` directory; please answer in the latex
- Grading is anonymous (I use a local script) (This also means you shouldn't put your name in your code or latex)

Reminders

- OK to collaborate on the main code assignment; remember to cite if you do!!
- Other questions (labelled "Problem" in the handout) should be done on your own with only high-level collaboration
- Assignments are meant to challenge everyone across a fairly broad set of areas!
  - May be particularly difficult when not in your wheelhouse
  - Talk to me; remember that you're not expected to get 100% on every assignment

Handing in assignments document

Accessing lab computers

- Ssh into one of the listed lab computers, or work in person
- Use software you're comfortable with. But I'm happy to help if you're learning something. (vim and/or emacs are something you should all probably know the basics of at some point.)
  - You should almost certainly use software with syntax highlighting and some automatic indentation. Nowadays it's best if you use software with an LSP (or something similar) that tells you about errors immediately.
  - Modern tools are personal in terms of specifics, but almost give very significant improvements in output
- Working in person on lab machines mean you can use a GUI application

Accessing lab computers from off campus

- Need to use a VPN. (OIT recommends Cisco. Consider `openconnect` as an alternative as Cisco is becoming insistent on putting garbage on your computer)

- Cannot ssh into the lab computers directly!
  - First ssh into `lohani.` From lohani, ssh into lab computers.
  - This double-ssh is a bit of a pain, and may cause issues with things like syntax highlighting.
  - Working on your computer; pushing and running on lab computers; is always an option.

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 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 $\frac{1}{2}\sum_{s \in S} s$.

- 136 method: try all subsets. For each, calculate its height; store best seen at each point.

- Running time? Space?

Some implementation details

- Can store a subset using an int of at most n bits (all instances have $n \leq 64$)
- 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.
- There must be SOME subset of S_1 in the correct final smaller tower.

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)
```

- Running time?

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?
- Answer: sort all subsets of S_2. Then can answer this query using binary search!

Meet in the Middle

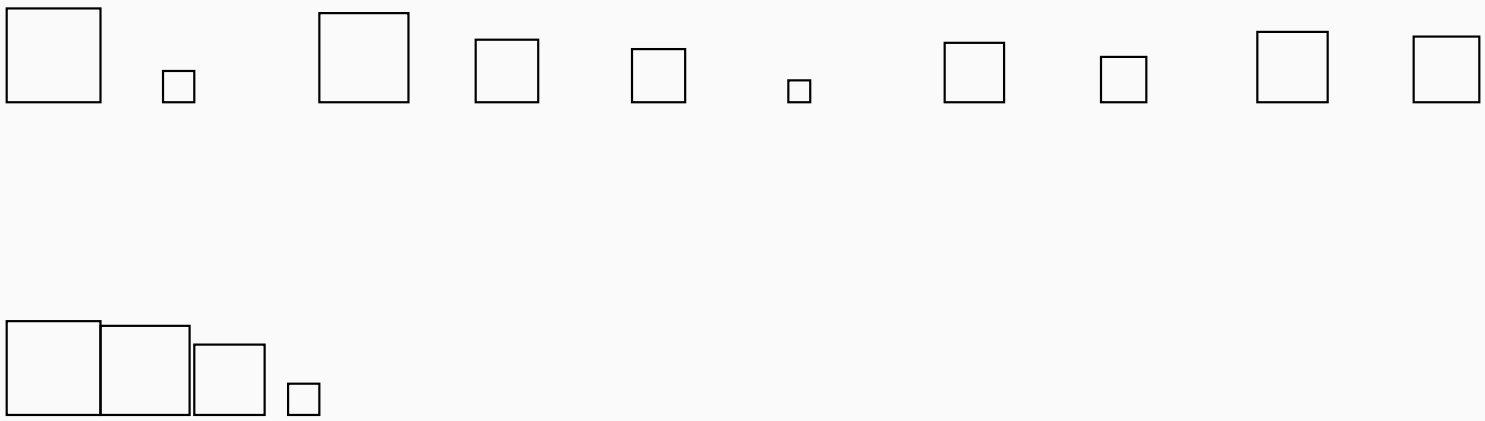
```
Fill array P with all
    subsets of S_2
Sort P by height
for each subset A_1 of S_1:
   s_1 ← h(A_1)
   binsearch(P, h(S)/2 − s_1)
   updateMax(h(A_2) + s_1)
```

- 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

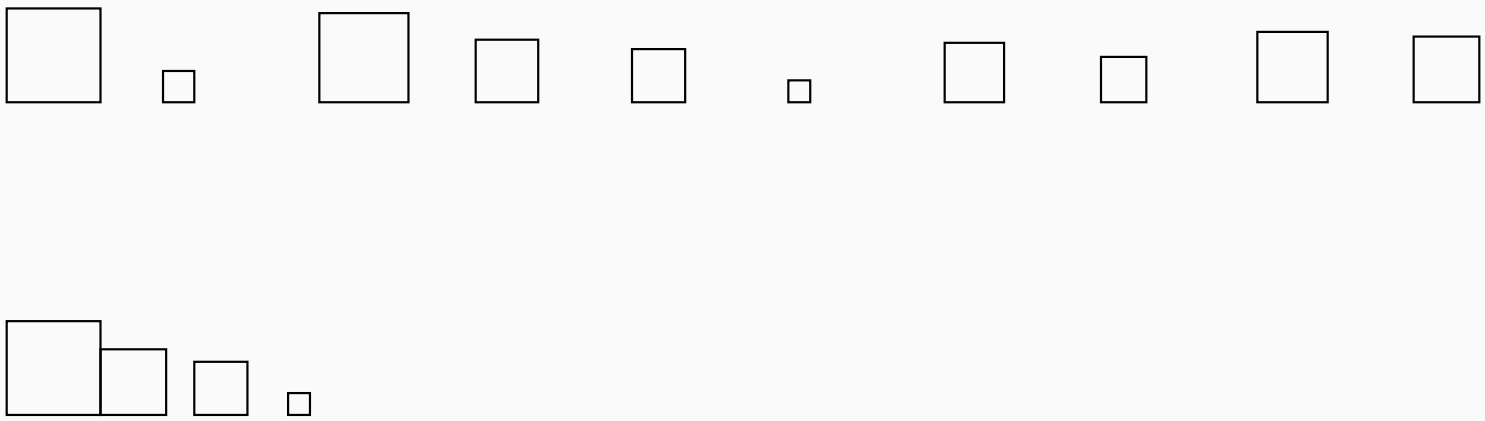
- Before we go forward, let's go over the high level strategy

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 subsets.

Partition the blocks into two equal-sized subsets. 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.

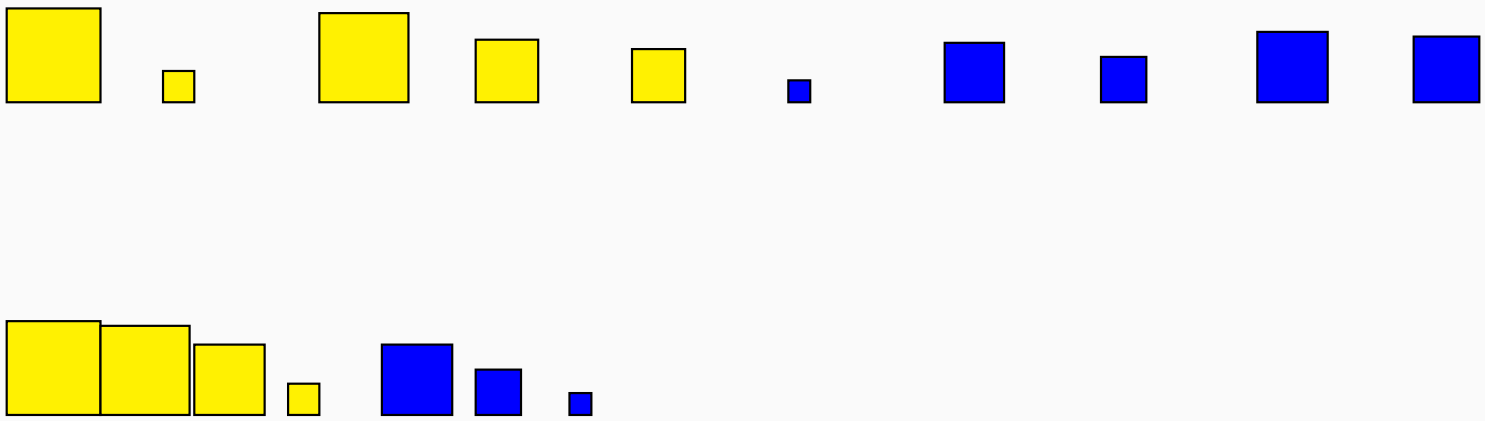
Meet in the Middle

0.0	00000
5.1	00010
7.2	00001
9.8	00100

...

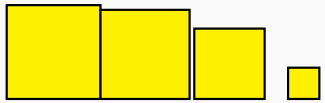
First, let's do some brute force preprocessing on the blue blocks. Go through all subsets of the blue blocks, and store their heights in a table. Then, sort the table by height.

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

- $O(n2^n)$ space, $O(n2^n)$ 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?
- $O(n2^{n/2})$ space is a lot. Is this worth it?
- Wait, can we do better than this?

Some questions about meet in the middle

- How 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. 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?

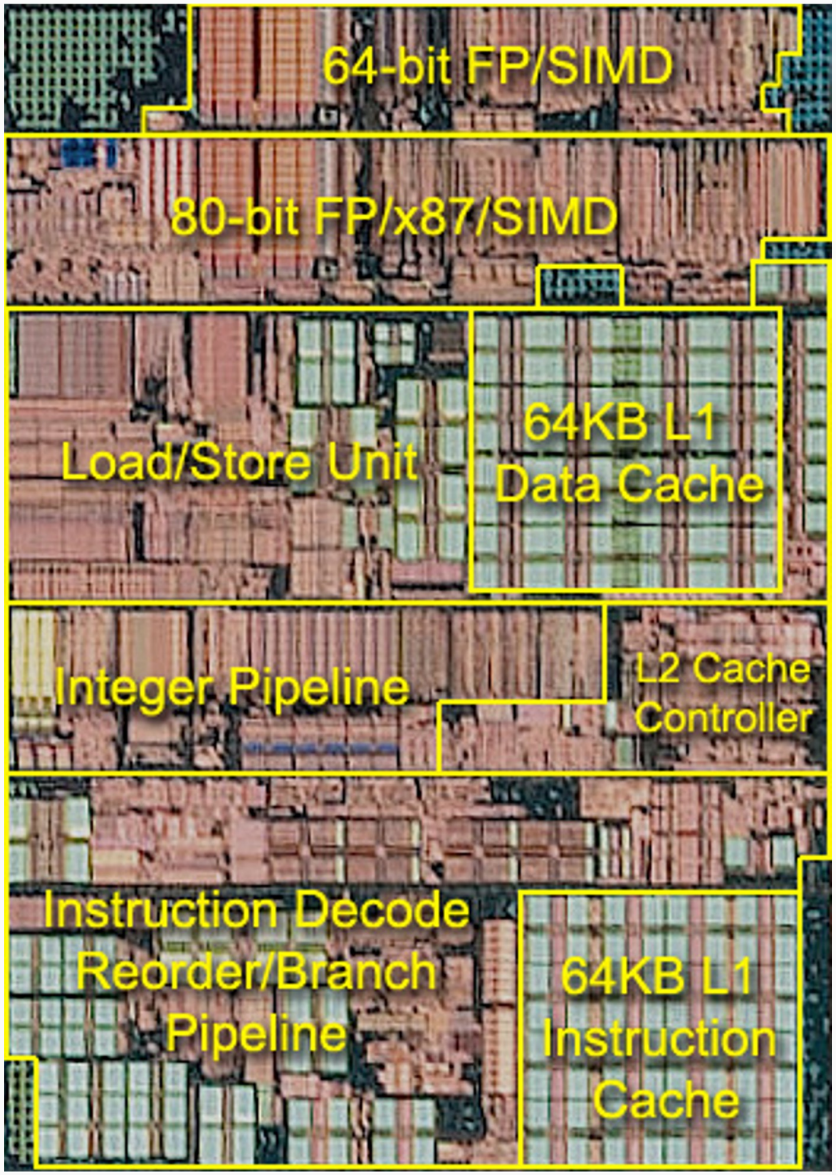
Any lingering questions about Assignment 1 or MITM?

Optimization Continued

More complicated operations

- Square root?
  - fast on our machines! 1-2 cycles
- memory allocation in bytes?
  - Pretty slow...
- memory allocation in megabytes?
- how does it grow as we increase the number of operations?
  - Cache efficiency is the problem here, not the memory call itself
  - (For what it's worth: `malloc` really is $O(1)$)

Modern processors



- Lots going on
- Moving things around takes more time than processing

Moving data around

- Casts can be expensive if they require moving the data into another part of the processor!

- (Can be free if they don't)

Branch mispredictions, etc.

- Instructions need to be moved into the CPU
- Modern CPUs predict what instructions will be next; move while completing other operations
- What if the CPU gets it wrong?
- "Branch misprediction:" 10-20 cycles to fetch the new instructions from memory
- Can have similar issues with calling non-inlined functions (compiler is very good at avoiding this)

Branch predictors

- CPU keeps track of your branches as it runs
  - Divides into four categories of how likely it is to be taken
- `gcc` also predicts your branches during compilation
- Can also give preprocessor directives about branches. Can be helpful (one of the last things you should do for optimization)

Avoiding branch mispredictions

```
int max(int a, int b) {
   int diff = a - b;
   int dsgn = diff >> 31;
   return a - (diff & dsgn);
}
```

```
int swap(int a, int b) {
   a = a ^ b;
   b = a ^ b;
   a = a ^ b;
}
```

- Avoid branches (ifs, etc.)
- (Crazy tricks often not worth it nowadays)
- `cmov` operations help a lot in modern processors; compilers are great at avoiding expensive branches
- If you do create a branch, ask yourself how easy it is to predict!
- Only way to be sure is to experiment

Profilers examples: gprof

- Compile with `-pg` option; then run normally; then run gprof on the executable
- Gives information about what calls what and how much time is in each
- Not perfect, but gives us some information, especially for simpler programs
  - Can see if one function is called a LOT
  - Can see if one function is only ever called by one other function
- Gets confusing with recursive calls
- I may ask you to use this, but be aware that it's useful sometimes at best

Profilers examples: cachegrind

- Compile with debugging info on `-g` AND optimizations on
  - What does this entail immediately?
- Then `valgrind --tool=cachegrind [your program]`
- Use `--branch-sim=yes` for branch prediction statistics.
- Outputs number of cache misses for instructions, then data, then combined
- Simulates a simple cache (based on your machine) with separate L1 caches for instructions and data, and unified L2 and (if on machine) L3 caches
- Does L1 misses vs last level (L3) misses

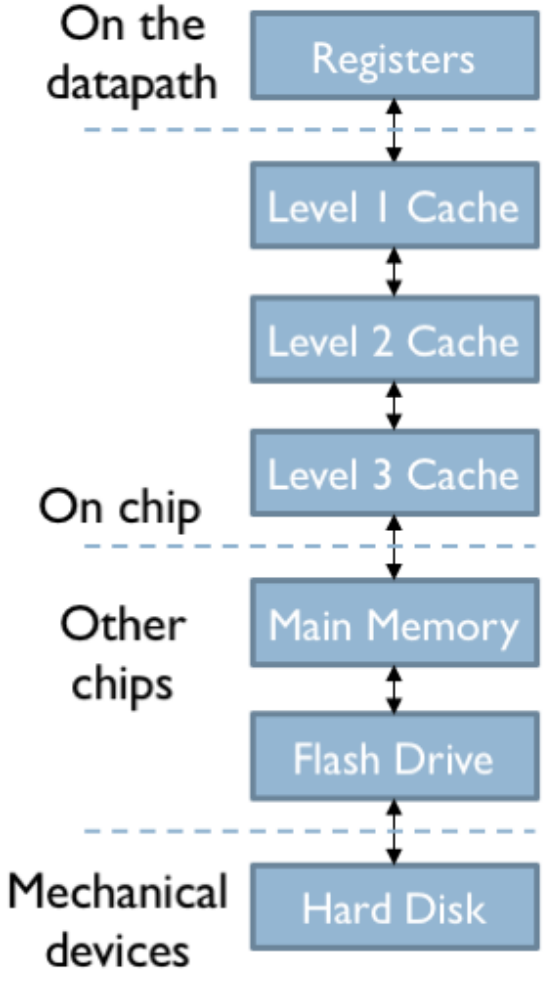
Final major cost: cache misses!

- Data is stored in different places on the computer

- Cost to access it frequently dominates running time

A Typical Memory Hierarchy

- Everything is a cache for something else…



Access time	Capacity	Managed By
1 cycle	1 KB	Software/Compiler
2-4 cycles	32 KB	Hardware
10 cycles	256 KB	Hardware
40 cycles	10 MB	Hardware
200 cycles	10 GB	Software/OS
10-100us	100 GB	Software/OS
10ms	1 TB	Software/OS

How caches work

- Stores data in the optimal(ish) place
- Moves data around in *cache lines* of 64 bytes
- Modern caches are very complicated
- Can be advantages of adjacent cache lines
- Basically: close is good; jumping around is bad.

Optimization Conclusions

- Different places where we can incur costs:
  - Operations
  - Branches and moving around instructions
  - Cache misses
- Determining costs is a matter of experimentation on modern machines!
  - Rarely perfect!
- Theme throughout class: design different experiments to test different aspects of code performance.

If we have time: MitM example