Contents

- Kruskal’s algorithm
- Union Find
- Integer multiplication
Back to MSTs: Another MST algorithm

- Kruskal’s MST:
  - Insert all edges into a PQ
  - Grab the min edge from the PQ that does not create a cycle in the MST
  - Remove it from the PQ and add it to the MST
Kruskal's example

PQ:

1: (0, 2)
2: (3, 5)
3: (1, 4)
4: (2, 5)
5: (2, 3)
5: (0, 3)
5: (1, 2)
6: (0, 1)
6: (2, 4)
6: (4, 5)
Kruskal’s runtime

- Instead of building up the MST starting from a single vertex, we build it up using edges all over the graph.
- How do we efficiently implement cycle detection?
Weighted tree example
public UF(int n) {
    count = n;
    id = new int[n];
    sz = new int[n];
    for (int i = 0; i < n; i++) { id[i] = i; sz[i] = 1; }
}

public void union(int p, int q) {
    int i = find(p), j = find(q);
    if (i == j) return;
    if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
    else               { id[j] = i; sz[i] += sz[j]; }
    count--;       
}
Weighted tree approach analysis

- Runtime?
  - find():
    - $\Theta(\log n)$
  - union():
    - $\Theta(\log n)$

- Can we do any better?
With this knowledge of union/find, how, exactly can it be used as a part of Kruskal’s algorithm?

- What is the runtime of Kruskal’s algorithm?
  - It’s $O(E \log E)$, because we need to sort the edges by weight first.
Gradeschool algorithm on binary numbers

\[
\begin{array}{c}
10100000100 \\
x 101100100 \\
\hline
00000000000 \\
00000000000 \\
101000010000 \\
00000000000000 \\
00000000000000 \\
10100001000000 \\
1010000100000000 \\
0000000000000000 \\
101000010000000000 \\
10100001000000000000 \\
000000000000000000 \\
1010000100000000000000 \\
\hline
1101111100110010000
\end{array}
\]
Let’s try to divide and conquer:

- Break our n-bit integers in half:
  - \( x = 1001011011001000, n = 16 \)
  - Let the high-order bits be \( x_H = 10010110 \)
  - Let the low-order bits be \( x_L = 11001000 \)
  - \( x = 2^{n/2}x_H + x_L \)
  - Do the same for \( y \)
  - \( x \times y = (2^{n/2}x_H + x_L) \times (2^{n/2}y_H + y_L) \)
  - \( x \times y = 2^n x_H y_H + 2^{n/2} (x_H y_L + x_L y_H) + x_L y_L \)
So what does this mean?

4 multiplications of n/2 bit integers

3 additions of n-bit integers

\[ 2^n x_H y_H + 2^{n/2} (x_H y_L + x_L y_H) + x_L y_L \]

A couple shifts of up to n positions

Actually 16 multiplications of n/4 bit integers (plus additions/shifts)

Actually 64 multiplications of n/8 bit integers (plus additions/shifts)

...
● Recursion really complicates our analysis…

● We’ll use a recurrence relation to analyze the recursive runtime

○ Goal is to determine:
  ■ How much work is done in the current recursive call?
  ■ How much work is passed on to future recursive calls?
  ■ All in terms of input size
Assuming we cut integers exactly in half at each call
  ○ I.e., input bit lengths are a power of 2

Work in the current call:
  ○ Shifts and additions are $\Theta(n)$

Work left to future calls:
  ○ 4 more multiplications on half of the input size

$T(n) = 4T(n/2) + \Theta(n)$
Soooo... what’s the runtime?

- Need to solve the recurrence relation
  - Remove the recursive component and express it purely in terms of \( n \)
    - A “cookbook” approach to solving recurrence relations:
      - The master theorem
The master theorem

- Usable on recurrence relations of the following form:

  \[ T(n) = aT\left(\frac{n}{b}\right) + f(n) \]

- Where:
  - \( a \) is a constant \( \geq 1 \)
  - \( b \) is a constant \( > 1 \)
  - and \( f(n) \) is an asymptotically positive function
Applying the master theorem

\[ T(n) = aT(n/b) + f(n) \]

- If \( f(n) \) is \( O(n^{\log_b(a) - \varepsilon}) \):
  - \( T(n) \) is \( \Theta(n^{\log_b(a)}) \)

- If \( f(n) \) is \( \Theta(n^{\log_b(a)}) \)
  - \( T(n) \) is \( \Theta(n^{\log_b(a)} \lg n) \)

- If \( f(n) \) is \( \Omega(n^{\log_b(a) + \varepsilon}) \) and \( (a \cdot f(n/b) \leq c \cdot f(n)) \) for some \( c < 1 \):
  - \( T(n) \) is \( \Theta(f(n)) \)
For our divide and conquer approach

\[ T(n) = 4T(n/2) + \Theta(n) \]

- \( a = 4 \)
- \( b = 2 \)
- \( f(n) \) is \( \Theta(n) \)
- So...
  - \( n^{\log_b a} = n^{\log_2 4} = n^2 \)
  - Being \( \Theta(n) \) means \( f(n) \) is polynomially smaller than \( n^2 \)
  - \( T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 4}) = \Theta(n^2) \)

- If \( f(n) \) is \( O(n^{\log_b a} - \epsilon) \):
  - \( T(n) \) is \( \Theta(n^{\log_b a}) \)

- If \( f(n) \) is \( \Theta(n^{\log_b a}) \):
  - \( T(n) \) is \( \Theta(n^{\log_b a} \lg n) \)

- If \( f(n) \) is \( \Omega(n^{\log_b a} + \epsilon) \) and \( (a \cdot f(n/b) \leq c \cdot f(n)) \) for some \( c < 1 \):
  - \( T(n) \) is \( \Theta(f(n)) \)
Leaves us back where we started with the grade school algorithm…

- Actually, the overhead of doing all of the dividing and conquering will make it slower than grade school
Let’s look for a smarter way to divide and conquer
Look at the recurrence relation again to see where we can improve our runtime:

$$T(n) = 4T(n/2) + \Theta(n)$$

Can we reduce the amount of work done by the current call?
Can we reduce the number of subproblems?
Can we reduce the subproblem size?
Karatsuba’s algorithm

- By reducing the number of recursive calls (subproblems), we can improve the runtime

\[ x \times y = 2^n x_H y_H + 2^{n/2} (x_H y_L + x_L y_H) + x_L y_L \]

M1 M2 M3 M4

- We don’t actually need to do both M2 and M3
  - We just need the sum of M2 and M3
    - If we can find this sum using only 1 multiplication, we decrease the number of recursive calls and hence improve our runtime
Karatsuba craziness

- $M_1 = x_h y_h$; $M_2 = x_h y_l$; $M_3 = x_l y_h$; $M_4 = x_l y_l$
- The sum of all of them can be expressed as a single mult:
  - $M_1 + M_2 + M_3 + M_4$
  - $= x_h y_h + x_h y_l + x_l y_h + x_l y_l$
  - $= (x_h + x_l) * (y_h + y_l)$
- Let's call this single multiplication $M_5$:
  - $M_5 = (x_h + x_l) * (y_h + y_l) = M_1 + M_2 + M_3 + M_4$
- Hence, $M_5 - M_1 - M_4 = M_2 + M_3$
- So: $x * y = 2^n M_1 + 2^{n/2}(M_5 - M_1 - M_4) + M_4$
  - Only 3 multiplications required!
  - At the cost of 2 more additions, and 2 subtractions
To get M5, we have to multiply (at most) \( n/2 + 1 \) bit ints
  - Asymptotically the same as our other recursive calls
 Requires extra additions and subtractions…
  - But these are all \( \Theta(n) \)
 So, the recurrence relation for Karatsuba’s algorithm is:
  - \( T(n) = 3T(n/2) + \Theta(n) \)
    ■ Which solves to be \( \Theta(n^{\lg 3}) \)
      - Asymptotic improvement over grade school algorithm!
        - For large \( n \), this will translate into practical improvement
Large integer multiplication in practice

- Can use a hybrid algorithm of grade school for large operands, Karatsuba’s algorithm for VERY large operands
  - Why are we still bothering with grade school at all?
The Schönhage–Strassen algorithm
- Uses Fast Fourier transforms to achieve better asymptotic runtime
  - $O(n \log n \log \log n)$
  - Fastest asymptotic runtime known from 1971-2007
- Required $n$ to be astronomical to achieve practical improvements to runtime
  - Numbers beyond $2^{2^{15}}$ to $2^{2^{17}}$

Fürer was able to achieve even better asymptotic runtime in 2007
- $n \log n 2^{O(\log^* n)}$
- No practical difference for realistic values of $n$