MA/CSSE 473 Day 04

Multiplication runtime

Multiplication based on Gauss formula

Mathematical induction review

MA/CSSE 473 Day 04

- Divide and Conquer Multiplication à la Gauss
- Mathematical Induction review
- Tiling with Trominoes (if there is time)
  - http://www3.amherst.edu/~nstarr/trom/puzzle-8by8/

What questions do you have?

EFFICIENT INTEGER MULTIPLICATION CONTINUED

For reference: The Master Theorem

- The Master Theorem for Divide and Conquer recurrence relations:
- Consider the recurrence $T(n) = aT(n/b) + f(n)$, $T(1)=c$, where $f(n) = \Theta(n^k)$ and $k \geq 0$,
- The solution is
  - $\Theta(n^k)$ if $a < b^k$
  - $\Theta(n^k \log n)$ if $a = b^k$
  - $\Theta(n^{\log_b a})$ if $a > b^k$

For details, see Levitin pages 483-485 or Weiss section 7.5.3.

Grimaldi's Theorem 10.1 is a special case of the Master Theorem.

We will use this theorem often. You should review its proof soon (Weiss's proof is a bit easier than Levitin's).

Recap: "Ordinary" Multiplication

- Example: multiply 13 by 11

```
            1   1   0   1
        x   1   0   1   1
-----------------------------
            1   1   0   1     (1101 times 1)
        1   1   0   1         (1101 times 1, shifted once)
    0   0   0   0             (1101 times 1, shifted twice)
1   1   0   1                 (1101 times 1, shifted thrice)
-----------------------------
1   0   0   0   1   1   1   1     (binary 143)
```

- There are n rows of 2n bits to add, so we do an O(n) operation n times, thus the whole multiplication is O( ) ?
- Can we do better?

New Multiplication Approach

- **Divide and Conquer**
- To multiply two n-bit integers x and y:
  - Split each into its left and right halves so that
    $x = 2^{n/2}x_L + x_R$, and $y = 2^{n/2}y_L + y_R$
  - The straightforward calculation of xy would be
    $(2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) =$
    $2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$
  - Code on next slide
  - Thus T(n) = . Solution?

Divide and Conquer multiplication

```
def multiply(x, y, n):
    """multiply two integers x and y, where n >= 0 is a power of 2,
       and is >= the maximum number of bits in x or y"""

    if n == 1:
        return x * y

    n_over_two = n//2

    two_to_the_n_over_two = 1 << n_over_two # a right bit-shift

    # note: these 2 operations can be done by bit shifts and masking.
    xL, xR = x // two_to_the_n_over_two, x % two_to_the_n_over_two
    yL, yR = y // two_to_the_n_over_two, y % two_to_the_n_over_two

    p1 = multiply (xL, yL, n_over_two)
    p2 = multiply (xL, yR, n_over_two)
    p3 = multiply (xR, yL, n_over_two)
    p4 = multiply (xR, yR, n_over_two)

    return (p1 << n) + ((p2 + p3) << n_over_two) + p4

print multiply((3000, 40000, 16))
```

Recurrence relation: solution:

Can we do better than O(n^2)?

- Is there an algorithm for multiplying two n-bit numbers in time that is less than O(n^2)?
- **Basis:** A discovery of Carl Gauss (1777-1855)
  - Multiplying complex numbers:
  - **(a + bi)(c+di) = ac – bd + (bc + ad)i**

Gauss's Algorithm

- **(a + bi)(c+di) = ac – bd + (bc + ad)i**
  - Needs **four** real-number multiplications and **three** additions
- But **bc + ad = (a+b)(c+d) – ac –bd**
  - And we have already computed **ac** and **bd** when we computed the real part of the product!
- Thus we can do the complex product with three multiplications and five additions
- Additions are so much faster than multiplications that we can essentially ignore them.
- A little savings, but not a big deal until applied recursively!

Code for Gauss-based Algorithm

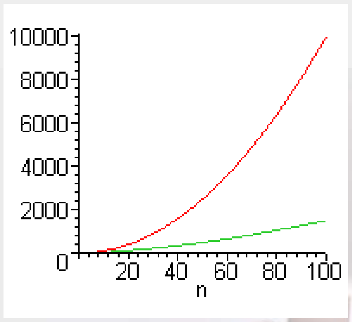
```
def multiply(x, y, n):
    """multiply two integers x and y, where n >= 0
       is a power of 2, and as large as the maximum number of bits in x or y"""

    if n == 1:
        return x * y

    n_over_two = n // 2  # simply shifts the bits one to the right.

    two_to_the_n_over_two = 1 << n_over_two


    xL, xR = x // two_to_the_n_over_two, x % two_to_the_n_over_two
    yL, yR = y // two_to_the_n_over_two, y % two_to_the_n_over_two
    # note that these two operations could be done by bit shifts and masking.

    p1 = multiply (xL,    yL,    n_over_two)
    p2 = multiply (xL+xR, yL+yR, n_over_two)
    p3 = multiply (xR,    yR,    n_over_two)


    return (p1 << n) + ((p2 - p3 - p1) << n_over_two) + p3
```

Recurrence relation: solution:

Is this really a lot faster?

- Standard multiplication: $\Theta(n^2)$
- Divide and conquer with Gauss trick: $\Theta(n^{1.59})$
- But there is a lot of additional overhead with Gauss, so standard multiplication is faster for small values of n.

```
plot( {n^2, n^1.59}, n=0..100);
```



- In reality we would not let the recursion go down to the single bit level, but only down to the number of bits that our machine can multiply in hardware without overflow.

Back to the "review thread"

QUICK REVIEW OF MATHEMATICAL INDUCTION

Induction Review

- To show that property* P(n) is true for all integers $n \geq n_0$, it suffices to show:
  - **Ordinary Induction**
    - $P(n_0)$ is true
    - For all $k \geq n_0$, if P (k) is true, then P(k+1) is also true.

  or
  - **Strong Induction**
    - $P(n_0)$ is true (sometimes you need multiple base cases)
    - For all $k > n_0$, if P(j) is true for all j with $n_0 \leq j < k$, then P(k) is also true.

* In this context, a **property** is a function whose domain is a subset of the non-negative integers and whose range is {true, false}

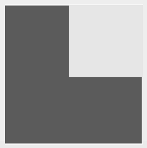
Induction examples

- For all N≥0, $\displaystyle\sum_{i=1}^{N} i \cdot 2^i = 2^{N+1}(N-1)+2$
  - This is formula 7 on P 470,
- Show that any postage amount of 24 cents or more can be achieved using only 5-cent stamps and 7-cent stamps.

Another Induction Example

Tiling with Trominoes

- We saw that a $2^n \times 2^n$ checkerboard can be tiled with dominoes.
- What about trominoes?
- Clearly, we can't tile an entire board!

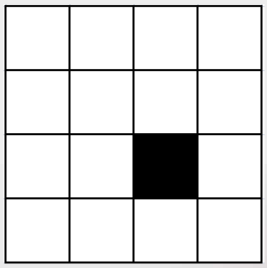


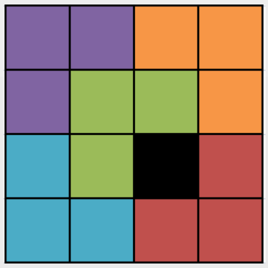
- **Definition:** A **deficient** rectangular grid of squares is one that has one square missing.
- It's easy to see that we can tile any 2×2 deficient rectangle! (We can rotate the tromino)

Note: HW 4 is about tiling with trominoes.

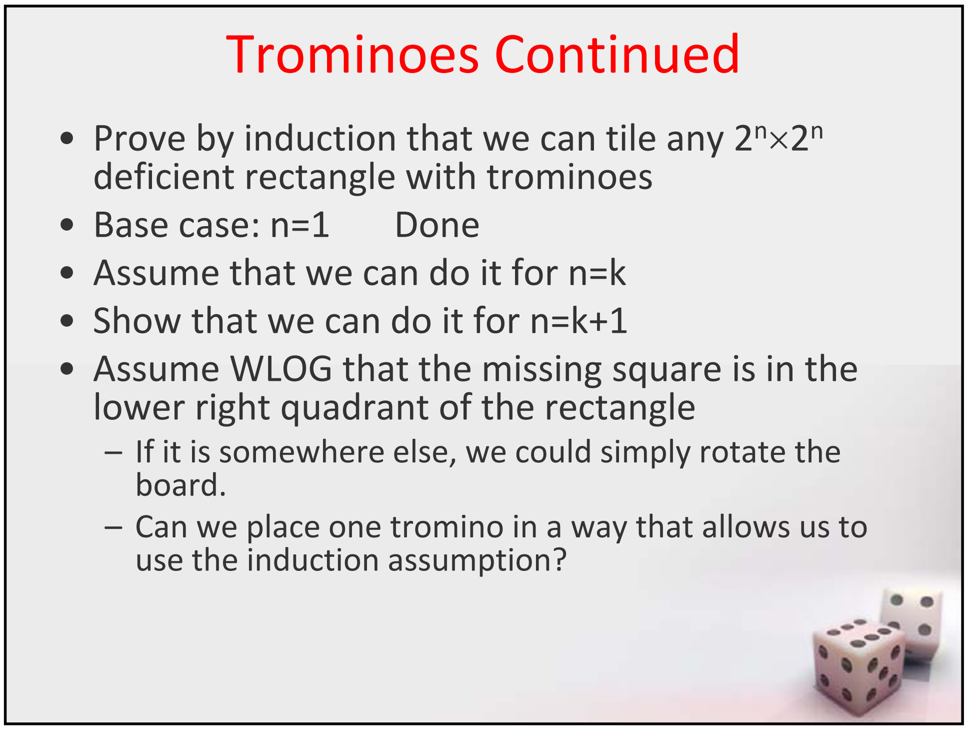
Trominoes Continued

- What about a 4 x 4 deficient rectangle?
- Can we tile this?



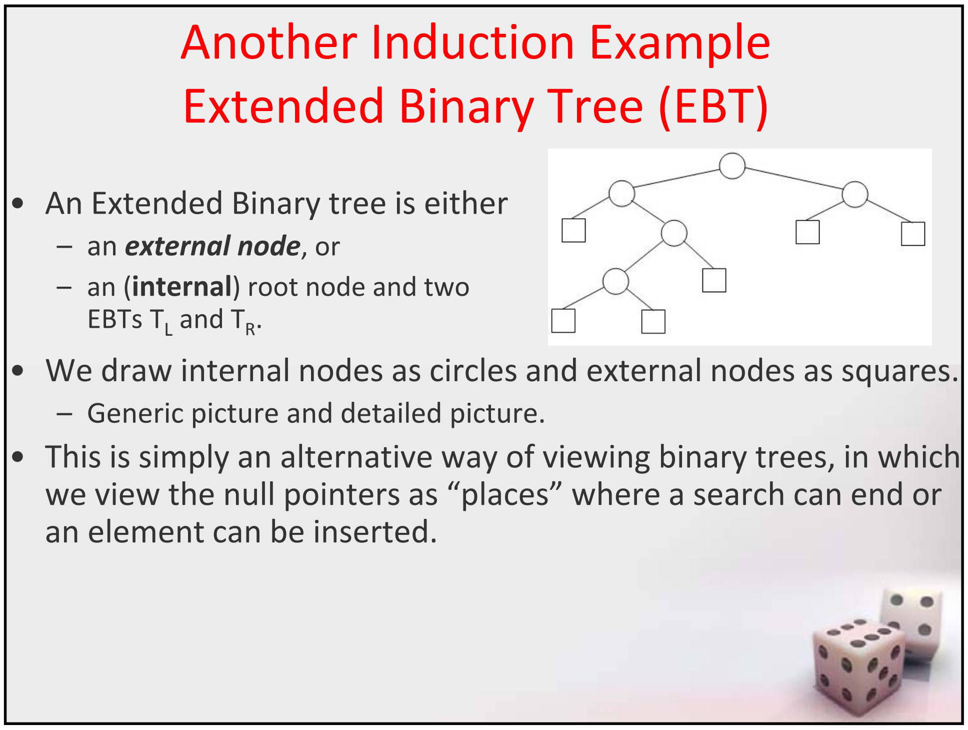


Fun with Tromino tiling:
http://www3.amherst.edu/~nstarr/trom/puzzle-8by8/



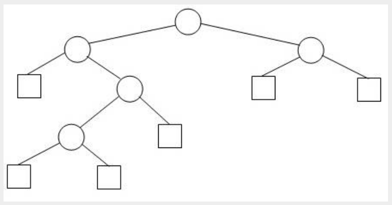
Trominoes Continued

- Prove by induction that we can tile any $2^n \times 2^n$ deficient rectangle with trominoes
- Base case: n=1 Done
- Assume that we can do it for n=k
- Show that we can do it for n=k+1
- Assume WLOG that the missing square is in the lower right quadrant of the rectangle
  - If it is somewhere else, we could simply rotate the board.
  - Can we place one tromino in a way that allows us to use the induction assumption?



Another Induction Example
Extended Binary Tree (EBT)

- An Extended Binary tree is either
  - an ***external node***, or
  - an (**internal**) root node and two EBTs T_L and T_R.



- We draw internal nodes as circles and external nodes as squares.
  - Generic picture and detailed picture.
- This is simply an alternative way of viewing binary trees, in which we view the null pointers as “places” where a search can end or an element can be inserted.

A property of EBTs

- **Property** P(N): For any N>=0, any EBT with N internal nodes has ________ external nodes.
- **Proof by strong induction**, based on the recursive definition.
  - A notation for this problem: IN(T), EN(T)
  - Note that, like some other simple examples, this one can be done without induction.
  - But the purpose of this exercise is practice with strong induction, especially on binary trees.
- What is the crux of any induction proof?
  - Finding a way to relate the properties for larger values (in this case larger trees) to the property for smaller values (smaller trees). **Do the proof now**.