In [1]:

```python
# setup
from IPython.core.display import display,HTML
display(HTML('<style>.prompt{width: 0px; min-width: 0px; visibility: collapse}</style>'))
display(HTML(open('rise.css').read()))

# imports
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
sns.set(style="whitegrid", font_scale=1.5, rc={'figure.figsize':(12, 6)})
import time
```

# CMPS 2200

# Introduction to Algorithms

## Recurrences - Example Recursive Algorithm

### Summary of last lecture

**Brick method:**

- **Root dominated**: cost *decreases* geometrically as we descend the tree
  - $W(n) \in O(\text{root})$
  - e.g., $W(n) = 2W(\frac{n}{2}) + n^2$
- **Leaf dominated**: cost *increases* geometrically as we descend the tree
  - $W(n) \in O(\text{leaves})$
  - e.g., $W(n) = 2W(\frac{n}{2}) + \sqrt{n}$
- **Balanced**: neither of the above is true
  - $W(n) \in O((\text{num levels}) * (\text{max cost at any level}))$
  - e.g., $W(n) = W(n-1) + n \in O(n^2)$
  - e.g., $W(n) = 2W(\frac{n}{2}) + n \in O(n \lg n)$

### Divide and Conquer

Even though we haven't explicitly given it a name, many of the algorithms we've been discussing are divide-and-conquer algorithms

- Split Sum
- Merge Sort
- Longest Sum Recursive

A divide-and-conquer algorithm, at each step, divides the problem into subproblems, solves each, then combines the results to arrive at the final solution.

Recurrences can easily be written and interpretted from the perspective of divide and conquer algorithms.

$$W(n) = \begin{cases} c_b, & \text{if } n = 1 \\ aW(\frac{n}{b}) + f(n), & \text{otherwise} \end{cases}$$

- $a$ is the branching factor.
- $\frac{n}{b}$ gives us the sub-problem size at the next level.
- $f(n)$ is the cost of the combining step.

## Integer multiplication

Now that we've come up with a general technique for expressing and solving recurrences, let's look at a recursive algorithm. You learned this algorithm in elementary school for integer multiplication:

- Input: $n$ bit integers $x = \langle x_{n-1}, \ldots, x_0 \rangle$ and $y = \langle y_{n-1}, \ldots, y_0 \rangle$
- Output: $x \cdot y$
- Example: '1001'$\times$'1101'

In [1]:
```python
def int2binary(n):
    return list('{0:b}'.format(n))
int2binary(9)
```

Out[1]:
```
['1', '0', '0', '1']
```

In [6]:
```python
nine = int2binary(9)
print(nine)
thirteen = int2binary(13)
print(thirteen)
```

```
['1', '0', '0', '1']
['1', '1', '0', '1']
```

```
         1001
       x 1101
       ======
         1001
        0000
       1001
    + 1001
    =========
      1110101   (117)
```

In [7]:
```python
def binary2int(n):
    return int(n, 2)
binary2int('1110101')
```

Out[7]:
```
117
```

What is the running time of the "elementary school" algorithm?

> For two $n$-digit inputs $O(n^2)$, since for each digit of $x$ we might add a stack of $n$ bits. The total number of bits in the solution is at most $2n$.

What does this have to do with recursion and recurrences?

Instead of the elementary school algorithm, consider splitting each $n$-digit input in half. Can we multiply recursively?

Let $x = \langle x_L, x_R \rangle$, $y = \langle y_L, y_R \rangle$. Then,

$$x = 2^{n/2} x_L + x_R \quad \text{e.g. } 1001 : 2^2(10) + (01) \tag{1}$$

$$y = 2^{n/2} y_L + y_R \quad \text{e.g. } 1101 : 2^2(11) + (01) \tag{2}$$

**Wait...Is multiplying by $2^{n/2}$ efficient?**

Recall: $2^2[10] \rightarrow [1000]$ (shift two places to the left).

So then,

$$x \cdot y = (2^{n/2} x_L + x_R)(2^{n/2} y_L + y_R) \tag{3}$$

$$= 2^n (x_L \cdot y_L) + 2^{n/2}(x_L \cdot y_R + x_R \cdot y_L) + (x_R \cdot y_R) \tag{4}$$

We've converted one multiplication of sizes $(n, n)$ into four multiplications of size $(\frac{n}{2}, \frac{n}{2})$.

What recursive algorithm, and recurrence is suggested by this observation?

> $W(n) = 4W(n/2) + cn$

What is the solution to this recurrence using the brick method? Is it root-dominated, or leaf-dominated?

## work of recursive multiplication

$C(\text{root}) = n$

$C(\text{level } 1) = 4(\frac{n}{2}) = 2 \cdot n$

geometrically **increasing** as we descend the recurrence tree.

A recurrence is **leaf-dominated** if for all $v$, there is an $\alpha > 1$ such that:

$$C(v) \leq \frac{1}{\alpha} \sum_{u \in D(v)} C(u)$$

let $\alpha \leftarrow 2$

$n \leq \frac{1}{2} \cdot 2 \cdot n$

> $W(n) = 4W(n/2) + cn$

The cost of a leaf dominated recurrence is $O(L)$, where $L$ is the number of leaves.

## how many leaf nodes are there?

nodes per level: $1, 4, 64, \ldots 4^i \ldots 4^{\lg n} = (2 \cdot 2)^{\lg n} = n \cdot n = n^2$

> This is a leaf-dominated reucrrence that is $O(n^2)$ -- the same as the elementary school algorithm!

Now, what is the span of this algorithm if implemented in parallel?

## span of recursive multiplication

Assuming each multiplication can be done in parallel, and that addition has span $O(n)$, we get that

$$S(n) = S(n/2) + cn$$

which yields a span of ...

**brick method**

$S(\text{root}) = n$

$S(\text{level 1}) = \frac{n}{2}$

$\rightarrow$ **root dominated**

$S(n) \in O(n)$. This is much better than the span of the grade school algorithm, which is $O(n^2)$!

**What parallelism is achieved?**

Parallelism $= \frac{W}{S} = \frac{n^2}{n} = n$

Can we do any better?

## recursive multiplication with less work

Recall that

$$\begin{aligned} x \cdot y = & \quad (2^{n/2} x_L + x_R)(2^{n/2} y_L + y_R) & (5) \\ = & \quad 2^n (x_L \cdot y_L) + 2^{n/2}(x_L \cdot y_R + x_R \cdot y_L) + (x_R \cdot y_R) & (6) \end{aligned}$$

Can we reduce this from 4 multiplications to 3??

Observation:

$$\begin{aligned} (x_L + x_R) \cdot (y_L + y_R) = & \quad (x_L \cdot y_L) + (x_L \cdot y_R) + (x_R \cdot y_L) + (x_R \cdot y_R) & (7) \\ (x_L \cdot y_R) + (x_R \cdot y_L) = & \quad (x_L + x_R) \cdot (y_L + y_R) - (x_L \cdot y_L) - (x_R \cdot y_R) & (8) \end{aligned}$$

How does our observation help us?

If we calculate $(x_L \cdot y_L)$, $(x_R \cdot y_R)$, and $(x_L + x_R) \cdot (y_L + y_R)$, that is *three* recursive multiplications.

So with 3 recursive multiplications and two more "additions", we then get that $W(n) = 3W(n/2) + dn$. What is the running time?

## work of $W(n) = 3W(n/2) + dn$

**brick method**

$C(\text{root}) = n$

$C(\text{level } 1) = \frac{3n}{2}$

$\frac{3}{2} > 1 \Rightarrow$ **leaf dominated**.

But, there are fewer leaves this time. Why?

nodes per level: $1, 3, 9, \ldots 3^i \ldots 3^{\lg n} = n^{\lg 3} \quad (\text{by } a^{\log_b c} = c^{\log_b a})$

Using the brick method, this is still a leaf-dominated recurrence, and thus the running time is $O(n^{\log_2 3})$ (approximately $O(n^{1.58})$ versus of $O(n^2)$ earlier).

This is known as the **Karatsaba-Ofman** algorithm (1962), and is the earliest known divide-and-conquer algorithm for integer multiplication. It is actually implemented in Intel hardware!

So, our we've decreased work from $O(n^2)$ to $O(n^{\lg 3})$.

Our span stays the same: $O(n)$   Why?

Parallelism $= \frac{W}{S} = \frac{n^{\lg 3}}{n} \approx n^{.58} < n$

So, our parallelism went down. Is that good or bad?

Schönhage and Strassen (1971) gave an algorithm that runs in $O(n \log n \log \log n)$ time.

In 2007, Fürer gave an algorithm that runs in $n \log n 2^{O(\log^* n)}$.

What is the fastest possible sequential algorithm for integer multiplication? In parallel?