Divide and Conquer

Algorithms
Divide and Conquer

• Generic recipe for many solutions:
  • Divide the problem into two or more smaller instances of the same problem
  • Conquer the smaller instances using recursion (or a base case)
  • Combine the answers to solve the original problem
**Integer Multiplication**

- Assume we want to multiply two $n$-bit integers with $n$ a power of two

- Divide: break the integers into two $n/2$-bit integers

\[
\begin{align*}
x &= 2^{\frac{n}{2}} x_L + x_R \\
y &= 2^{\frac{n}{2}} y_L + y_R
\end{align*}
\]
Integer Multiplication

- Conquer: Solve the problem of multiplying of $n/2$ bit integers by recursion or a base case for $n=1$, $n=2$, or $n=4$

\[
x = 2^{\frac{n}{2}}x_L + x_R
\]

\[
y = 2^{\frac{n}{2}}y_L + y_R
\]

\[
x_L \cdot y_L \quad x_L \cdot y_R \quad x_R \cdot y_L \quad x_R \cdot y_R
\]
Integer Multiplication

- Now combine:
  - In the naïve way:

\[
x \cdot y = (x_L \cdot 2^n + x_R) \cdot (y_L \cdot 2^n + y_R)
\]

\[
= x_L \cdot y_L \cdot 2^n + (x_L \cdot y_R + x_R \cdot y_L) \cdot 2^n + x_R \cdot y_R
\]
Integer Multiplication

\[ x \cdot y = (x_L2^{\frac{n}{2}} + x_R) \cdot (y_L2^{\frac{n}{2}} + y_R) \]

\[ = x_L \cdot y_L2^n + (x_L \cdot y_R + x_R \cdot y_L) \cdot 2^{\frac{n}{2}} + x_R \cdot y_R \]

- We count the number of multiplications
  - Multiplying by powers of 2 is just shifting, so they do not count
  - \( T(n) \) number of bit multiplications for integers with \( 2^n \) bits:
    - \( T(0) = 1 \)
    - Recursion:
      \[ T(n + 1) = 4T(n) \]
Integer Multiplication

- Solving the recursion
  \[ T(0) = 1 \]
  \[ T(n + 1) = 4T(n) \]

- Intuition:
  \[ T(n) = 4T(n - 1) = 4^2T(n - 2) = 4^3T(n - 3) = \ldots = 4^nT(0) = 4^n \]
Integer Multiplication

• Proposition: \( T(n) = 4^n \)

• Proof by induction:
  • Induction base:
    \[
    T(0) = 1 = 4^0
    \]
  • Induction step: Assume \( T(n) = 4^{n-1} \). Show \( T(n + 1) = 4^n \)
  • Proof:
    \[
    T(n) = 4T(n - 1) \text{ Recursion Equation}
    = 4 \times 4^{n-1} \text{ Induction Assumption}
    = 4^n
    \]
Integer Multiplication

- Since the number of bits is $m = 2^n$
- Number of multiplications is
  \[ S(m) = T(n) = 4^n = (2^n)^n = m^2 \]
- This is not better than normal multiplication
Integer Multiplication

• Now combine:

• Instead: 
  \[ x \cdot y = (x_L 2^n + x_R) \cdot (y_L 2^n + y_R) \]
  \[ = x_L \cdot y_L \cdot 2^n + (x_L \cdot y_R + x_R \cdot y_L) \cdot 2^{\frac{n}{2}} + x_R \cdot y_R \]

• Use \[ (x_L \cdot y_R + x_R \cdot y_L) = (x_L + x_R) \cdot (y_L + y_R) - x_L \cdot y_L - x_R \cdot y_R \]

• This reuses two multiplications that are already used
Integer Multiplication

- We need to deal with the potential overflow in calculating

\[(x_L + x_R) \cdot (y_L + y_R)\]
Integer Multiplication

- Now, we only do three multiplications of $2^n$ bit numbers in order to multiply two $2^{n+1}$ bit numbers.
- The recursion becomes

\[
T(0) = 1 \quad T(n + 1) = 3T(n)
\]
Integer Multiplication

- Solving the recurrence $T(0) = 1 \quad T(n + 1) = 3T(n)$
- Heuristics:

  $$T(n) = 3T(n - 1) = 3^2T(n - 2) = \ldots = 3^n T(0) = 3^n$$
Integer Multiplication

• As before prove exactly using induction
Integer Multiplication

The multiplication of two \( m = 2^n \)-bit numbers takes

\[
S(m) = T(n) \\
= 3^n \\
= 3^\log_2(m) \\
= \exp(\log(3^\log_2(m))) \\
= \exp(\log_2 m \log 3) \\
= \exp(\log m \log_3 \frac{1}{\log 2}) \\
= \exp(\log(m^{\log_2 3})) \\
= m^{\log_2 3}
\]
Integer Multiplication

- This way, multiplication of $m$-bit numbers takes $m^{1.58496}$ bit multiplications
Integer Multiplication

- Can be used for arbitrary length integer multiplication
- Base case is 32 or 64 bits
- But can still do better using Fast Fourier Transformation
Binary Search

• Given an array of ordered integers, a pointer to the beginning and to the end of a portion of the array, decide whether an element is in the slice

• $\text{Search}(\text{array}, \text{beg}, \text{end}, \text{element})$
Binary Search

• Divide: Determine the middle element. This divides the array into two subsets

• Conquer: Compare the element with the middle element. If it is smaller, find out whether the element is in the left half, otherwise, whether the element is in the right half

• Combine: Just return the answer to the one question
def binary_search(array, beg, end, key):
    if beg >= end:
        return False
    mid = (beg+end)//2
    if array[mid]==key:
        return True
    elif array[mid] > key:
        return binary_search(array, beg, mid, key)
    else:
        return binary_search(array, mid+1, end, key)

test = [2, 3, 5, 6, 12, 15, 17, 19, 21, 23, 27, 29, 31, 33, 35, 39, 41]
print(binary_search(test, 0, len(test), 21))
print(binary_search(test, 0, len(test), 22))
Binary Search

- Let $T(n)$ be the runtime of binary_search on a subarray with $n$ elements
- Recursion: There is a constant $c$ such that

\[ T(1) \leq c \]
\[ T(n) \leq T(n/2) + c \]
Binary Search

- Solving the recursion

\[ T(n) \leq T(n/2) + c \]
\[ \leq T(n/4) + 2c \]
\[ \ldots \]
\[ \leq T(n/2^m) + mc \]

- If \( m \geq \log_2 n \) then \( T(n) \leq T(1) + mc = (m + 1)c \)
Binary Search

- With other words, binary search on $n$ elements takes time
  \[ \propto \log_2(n) \]