Divide and Conquer

Stefan D. Bruda

CS 317, Fall 2024

Divide and Conquer

Idea:

1. If the problem is small enough, then solve it
2. Otherwise:
   1. Divide the problem into two or more sub-problems
   2. Solve each sub-problem recursively
   3. Combine the solutions to the sub-problems to obtain a solution to the original problem

Idea:

1. If the problem is small enough, then solve it
2. Otherwise:
   1. Divide the problem into two or more sub-problems
   2. Solve each sub-problem recursively
   3. Combine the solutions to the sub-problems to obtain a solution to the original problem

Example:

```
algorithm MergeSort(S, l, h):
    if l < h then
        m ← (l + h)/2                  // divide
        MergeSort(S, l, m)             // conquer
        MergeSort(S, m + 1, h)         // conquer
        Merge(S, l, m, h)              // combine
```

```
algorithm Merge(S, l, m, h):
    T ← ⟨⟩                             // merge placeholder
    i ← l                              // top of first half
    j ← m                              // top of second half
    k ← l                              // top of T
    while i ≤ m ∧ j ≤ h do
        if S_i < S_j then              // compare top
            T_k ← S_i                  // smaller in T
            i ← i + 1                  // advance top
        else
            T_k ← S_j                  // smaller in T
            i ← j + 1                  // advance top
        k ← k + 1
    while i ≤ m do                     // flush first half
        T_k ← S_i
        i ← i + 1
        k ← k + 1
    while j ≤ h do                     // flush second half
        T_k ← S_j
        j ← j + 1
        k ← k + 1
    for k = l to h do                  // result back into S
        S_k ← T_k
```

Lemma (correctness of MERGE)

If $S_{l\ldots m}$ and $S_{m+1\ldots h}$ are sorted then at the end of MERGE *the sequence $T_{l\ldots h}$ contains a sorted permutation of $S_{l\ldots h}$*

MERGESORT ANALYSIS

Lemma (correctness of MERGE)

If $S_{l\dots m}$ and $S_{m+1\dots h}$ are sorted then at the end of MERGE *the sequence $T_{l\dots h}$ contains a sorted permutation of $S_{l\dots h}$*

- Loop invariant (for all three loops): $T_{l\dots k-1}$ is sorted and contains exactly all the $k-1$ smallest elements of $S_{l\dots h}$
  - Proof by induction over k
- At the end of the loop $k = h+1$ and so the invariant implies the desired properties of T

Lemma (correctness of MERGE)

If $S_{l\ldots m}$ and $S_{m+1\ldots h}$ are sorted then at the end of MERGE *the sequence $T_{l\ldots h}$ contains a sorted permutation of $S_{l\ldots h}$*

- Loop invariant (for all three loops): $T_{l\ldots k-1}$ is sorted and contains exactly all the $k-1$ smallest elements of $S_{l\ldots h}$
  - Proof by induction over k
- At the end of the loop $k = h + 1$ and so the invariant implies the desired properties of T

Theorem (correctness of MERGESORT)

MERGESORT *replaces any input sequence $S_{h..l}$ with a sorted permutation of that sequence*

MERGESORT ANALYSIS

Lemma (correctness of MERGE)

If $S_{l...m}$ and $S_{m+1...h}$ are sorted then at the end of MERGE *the sequence $T_{l...h}$ contains a sorted permutation of $S_{l...h}$*

- Loop invariant (for all three loops): $T_{l...k-1}$ is sorted and contains exactly all the $k - 1$ smallest elements of $S_{l...h}$
  - Proof by induction over k
- At the end of the loop $k = h + 1$ and so the invariant implies the desired properties of T

Theorem (correctness of MERGESORT)

MERGESORT *replaces any input sequence $S_{h..l}$ with a sorted permutation of that sequence*

- Proof by induction on $h - l$:
  - In the base case $h - l = 0$ MERGESORT (correctly) does nothing
  - To sort $h - l$ values MERGESORT sorts correctly $(h - l)/2$ values two times (inductive hypothesis) and then correctly merges the two sub-sequences (lemma), thus obtaining a sorted permutation of the original sequence

MergeSort analysis (cont'd)

- $T(n) = 2T(n/2) + n$, $T(1) = 1$ so $T(n) = \Theta(n \log n)$ $\rightarrow$ already known!

MergeSort analysis (cont'd)

- $T(n) = 2T(n/2) + n$, $T(1) = 1$ so $T(n) = \Theta(n \log n)$ $\rightarrow$ already known!

Theorem (comparison sorting lower bound)

The lower bound for comparison sort algorithms is $\Omega(n \log n)$

MERGESORT ANALYSIS (CONT'D)

- $T(n) = 2T(n/2) + n$, $T(1) = 1$ so $T(n) = \Theta(n \log n)$ $\rightarrow$ already known!

Theorem (comparison sorting lower bound)

The lower bound for comparison sort algorithms is $\Omega(n \log n)$

- We count comparisons using a decision tree
  - Internal node $S_{i,j}$ represents a comparison between S_i and S_j
  - The left [right] sub-tree represents all the decisions to be made provided that $S_i \leq S_j$ [$S_i > S_j$]
  - Each leaf labeled with a different permutation of S
  - Following a path performs the sequence of comparison given by the sequence of nodes and produces the leaf permutation of S
- We have $n!$ permutations (leafs) so the minimum path from root to a leaf contains $\log(n!) = \Theta(n \log n)$ nodes
- So a sorting algorithm must perform $\Omega(n \log n)$ comparisons to differentiate between all the possible permutations

Corollary (optimality of MERGESORT)

MERGESORT *is optimal*

- Problem with MergeSort: require substantial extra space
- By contrast QuickSort is an in-place sorting algorithm

```
algorithm QuickSort(S, l, h):
    if l < h then
        Choose pivot S_x
        S_1 ↔ S_x
        p ← Partition(S, l, h)
        QuickSort(S, l, p − 1)
        QuickSort(S, p + 1, h)
```

```
algorithm Partition(S, l, h):        // ver. 1
    pivot ← S_l
    j ← l
    for i = l + 1 to h do
        if S_i < pivot then
            j ← j + 1
            S_i ↔ S_j
    S_l ↔ S_j
    return j
```

```
algorithm Partition(S, l, h):        // ver. 2
    pivot ← S_l
    i ← l
    j ← h + 1                        // start beyond ends
    repeat
        repeat i ← i + 1 until S_i > pivot:
        repeat j ← j − 1 until S_j < pivot:
        if i < j then S_i ↔ S_j
    until i > j:
    S_l ↔ S_j
     return j
```

Analysis of QuickSort

- Time complexity:
  - Best case: we always partition equally
    $T(n) = 2T(n/2) + n$, $T(1) = 1$ and so $T(n) = \Theta(n \log n)$
  - Worst case: one partition is always empty (when?)
    $T(n) = T(n-1) + n$, $T(1) = 1$ and so $T(n) = \Theta(n^2)$

Analysis of QuickSort

- Time complexity:
  - Best case: we always partition equally
    $T(n) = 2T(n/2) + n$, $T(1) = 1$ and so $T(n) = \Theta(n \log n)$
  - Worst case: one partition is always empty (when?)
    $T(n) = T(n-1) + n$, $T(1) = 1$ and so $T(n) = \Theta(n^2)$
  - Can mitigate (but not fix) the worst case by choosing the pivot randomly of the best out of k random values for a small constant k
- QuickSort is not stable

ANALYSIS OF QUICKSORT

- Time complexity:
  - Best case: we always partition equally
    $T(n) = 2T(n/2) + n$, $T(1) = 1$ and so $T(n) = \Theta(n \log n)$
  - Worst case: one partition is always empty (when?)
    $T(n) = T(n-1) + n$, $T(1) = 1$ and so $T(n) = \Theta(n^2)$
  - Can mitigate (but not fix) the worst case by choosing the pivot randomly of the best out of k random values for a small constant k
- QuickSort is not stable
- Correctness of PARTITION:
  - Loop invariant for version 1: At the end of an iteration all values $S_{l+1\ldots j}$ are smaller than *pivot* and no value $S_{j+1\ldots i}$ is smaller than *pivot*
  - Can verify by induction over i
  - Invariant implies desired postcondition that everything in $S_{l\ldots p-1}$ is less than *pivot* and nothing in $S_{p+1\ldots h}$ is less than the pivot
  - Loop invariant for version 2: At the end of an iteration all values in $S_{l+1\ldots i}$ are smaller than the pivot and no values in $S_{j\ldots h}$ are smaller than the pivot
  - Can verify by induction over the iteration number

Analysis of QuickSort

- Time complexity:
  - Best case: we always partition equally
    $T(n) = 2T(n/2) + n$, $T(1) = 1$ and so $T(n) = \Theta(n \log n)$
  - Worst case: one partition is always empty (when?)
    $T(n) = T(n-1) + n$, $T(1) = 1$ and so $T(n) = \Theta(n^2)$
  - Can mitigate (but not fix) the worst case by choosing the pivot randomly of the best out of k random values for a small constant k
- QuickSort is not stable
- Correctness of PARTITION:
  - Loop invariant for version 1: At the end of an iteration all values $S_{l+1\ldots j}$ are smaller than *pivot* and no value $S_{j+1\ldots i}$ is smaller than *pivot*
  - Can verify by induction over i
  - Invariant implies desired postcondition that everything in $S_{l\ldots p-1}$ is less than *pivot* and nothing in $S_{p+1\ldots h}$ is less than the pivot
  - Loop invariant for version 2: At the end of an iteration all values in $S_{l+1\ldots i}$ are smaller than the pivot and no values in $S_{j\ldots h}$ are smaller than the pivot
  - Can verify by induction over the iteration number
- Correctness of QUICKSORT: same as for MERGESORT (induction over $h - l$)

- We use the QuickSort idea to find the k-th smallest value in a given array, without sorting the array:

```
algorithm QuickSelect(k, S, l, h):
    if l < h then
        Choose pivot S_x
        S_1 ↔ S_x
        p ← Partition(S, l, h)
        if k = p then return S_k
        else if k < p then QuickSelect(k, S, l, p − 1)
        else QuickSelect(k, S, p + 1, h)
```

- We use the QuickSort idea to find the k-th smallest value in a given array, without sorting the array:

```
algorithm QuickSelect(k, S, l, h):
    if l < h then
        Choose pivot S_x
        S_1 ↔ S_x
        p ← Partition(S, l, h)
        if k = p then  return S_k
        else if k < p then  QuickSelect(k, S, l, p − 1)
        else  QuickSelect(k, S, p + 1, h)
```

- Correctness: just like for QuickSort
- Time complexity:
  - Best case: we always partition equally
    $T(n) = T(n/2) + n$, $T(1) = 1$ and so $T(n) = \Theta(n)$ (better than sorting)
  - Worst case: one partition is always empty
    $T(n) = T(n-1) + n$, $T(1) = 1$ and so $T(n) = \Theta(n^2)$

```
algorithm MOMSELECT(k, S, l, h):
    if h − l ≤ 25 then  use brute force
    else
        m ← (h − l)/5
        for i = 1 to m do
            M_i ← MEDIANOFFIVE(S_{l+5i−4...l+5i})    // brute force
            // Note: M can and should be an in-place array (within S)
        mom ← MOMSELECT(m/2, M, 1, m)
        S_1 ↔ S_mom
        p ← PARTITION(S, l, h)
        if k = p then  return S_k
        else if k < p then  MOMSELECT(k, S, l, p − 1)
        else  MOMSELECT(k, S, p + 1, h)
```

- Obviously correct (why?)

```
algorithm MoMSelect(k, S, l, h):
    if h − l ≤ 25 then use brute force
    else
        m ← (h − l)/5
        for i = 1 to m do
            M_i ← MedianOfFive(S_{l+5i−4...l+5i})    // brute force
            // Note: M can and should be an in-place array (within S)
        mom ← MoMSelect(m/2, M, 1, m)
        S_1 ↔ S_mom
        p ← Partition(S, l, h)
        if k = p then return S_k
        else if k < p then MoMSelect(k, S, l, p − 1)
        else MoMSelect(k, S, p + 1, h)
```

- Obviously correct (why?)
- *mom* is larger [smaller] than about $(h-l)/10$ block-of-five medians
- Each block median is larger [smaller] than 2 other elements in its block
- So *mom* is larger [smaller] than $3(h-l)/10$ elements in S and so cannot be farther than $7(h-l)/10$ elements from the perfect pivot
- So $T(n) = T(n/5) + T(7n/10) + n \Rightarrow T(n) = 10 \times c \times n \Rightarrow T(n) = \Theta(n)$

How to choose good pivots

```
algorithm MoMSelect(k, S, l, h):
    if h − l ≤ 25 then  use brute force
    else
        m ← (h − l)/5
        for i = 1 to m do
            M_i ← MedianOfFive(S_{l+5i−4...l+5i})    // brute force
            // Note: M can and should be an in-place array (within S)
        mom ← MoMSelect(m/2, M, 1, m)
        S_1 ↔ S_mom
        p ← Partition(S, l, h)
        if k = p then  return S_k
        else if k < p then  MoMSelect(k, S, l, p − 1)
        else  MoMSelect(k, S, p + 1, h)
```

- Obviously correct (why?)
- *mom* is larger [smaller] than about $(h - l)/10$ block-of-five medians
- Each block median is larger [smaller] than 2 other elements in its block
- So *mom* is larger [smaller] than $3(h - l)/10$ elements in S and so cannot be farther than $7(h - l)/10$ elements from the perfect pivot
- So $T(n) = T(n/5) + T(7n/10) + n \Rightarrow T(n) = 10 \times c \times n \Rightarrow T(n) = \Theta(n)$
    - Note in passing: $T(n) = T(n/3) + T(2n/3) + n \Rightarrow T(n) = \Theta(n \log n)$

```
algorithm MOMSELECT(k, S, l, h):
    if h − l ≤ 25 then  use brute force
    else
        m ← (h − l)/5
        for i = 1 to m do
            M_i ← MEDIANOFFIVE(S_{l+5i−4...l+5i})    // brute force
            // Note: M can and should be an in-place array (within S)
        mom ← MOMSELECT(m/2, M, 1, m)
        S_1 ↔ S_mom
        p ← PARTITION(S, l, h)
        if k = p then  return S_k
        else if k < p then  MOMSELECT(k, S, l, p − 1)
        else  MOMSELECT(k, S, p + 1, h)
```

- Obviously correct (why?)
- *mom* is larger [smaller] than about $(h-l)/10$ block-of-five medians
- Each block median is larger [smaller] than 2 other elements in its block
- So *mom* is larger [smaller] than $3(h-l)/10$ elements in S and so cannot be farther than $7(h-l)/10$ elements from the perfect pivot
- So $T(n) = T(n/5) + T(7n/10) + n \Rightarrow T(n) = 10 \times c \times n \Rightarrow T(n) = \Theta(n)$
    - Note in passing: $T(n) = T(n/3) + T(2n/3) + n \Rightarrow T(n) = \Theta(n \log n)$
- If QUICKSORT uses MOMSELECT to choose pivot then it gets down to $O(n \log n)$ worst-case complexity (optimal)

FAST MATRIX MULTIPLICATION

With A and B $n \times n$ matrices compute $C = A \times B$ such that $C_{i,j} = \sum_{k-1}^{n} A_{i,k} \times B_{k,j}$

- Straightforward algorithm of complexity $O(n^3)$
- Obvious lower bound $\Omega(n^2)$

With A and B $n \times n$ matrices compute $C = A \times B$ such that $C_{i,j} = \sum_{k-1}^{n} A_{i,k} \times B_{k,j}$

- Straightforward algorithm of complexity $O(n^3)$
- Obvious lower bound $\Omega(n^2)$
- Divide and conquer approach:

$$\left(\begin{array}{c|c} A_{\leftarrow\uparrow} & A_{\rightarrow\uparrow} \\ \hline A_{\leftarrow\downarrow} & A_{\rightarrow\downarrow} \end{array}\right) \times \left(\begin{array}{c|c} B_{\leftarrow\uparrow} & B_{\rightarrow\uparrow} \\ \hline B_{\leftarrow\downarrow} & B_{\rightarrow\downarrow} \end{array}\right) = \left(\begin{array}{c|c} C_{\leftarrow\uparrow} & C_{\rightarrow\uparrow} \\ \hline C_{\leftarrow\downarrow} & C_{\rightarrow\downarrow} \end{array}\right)$$

```
algorithm MatrixMul(n, A, B):
  if n = 2 then return A × B (brute force)
  else
    Partition A into A←↑, A→↑, A←↓, A→↓
    Partition B into B←↑, B→↑, B←↓, B→↓
    C←↑ ← MatrixMul(n/2, A←↑, B←↑)+MatrixMul(n/2, A→↑, B←↓)
    C→↑ ← MatrixMul(n/2, A←↑, B→↑)+MatrixMul(n/2, A→↑, B→↓)
    C←↓ ← MatrixMul(n/2, A←↓, B←↑)+MatrixMul(n/2, A→↓, B→↓)
    C→↓ ← MatrixMul(n/2, A←↓, B→↑)+MatrixMul(n/2, A→↓, B→↓)
    Combine C←↑, C→↑, C←↓, C→↓ into C
    return C
```

- $T(n) = 8T(n/2) + n^2$, $T(2) = 8$

With A and B $n \times n$ matrices compute $C = A \times B$ such that $C_{i,j} = \sum_{k-1}^{n} A_{i,k} \times B_{k,j}$

- Straightforward algorithm of complexity $O(n^3)$
- Obvious lower bound $\Omega(n^2)$
- Divide and conquer approach:

$$\left(\begin{array}{c|c} A_{\leftarrow\uparrow} & A_{\rightarrow\uparrow} \\ \hline A_{\leftarrow\downarrow} & A_{\rightarrow\downarrow} \end{array}\right) \times \left(\begin{array}{c|c} B_{\leftarrow\uparrow} & B_{\rightarrow\uparrow} \\ \hline B_{\leftarrow\downarrow} & B_{\rightarrow\downarrow} \end{array}\right) = \left(\begin{array}{c|c} C_{\leftarrow\uparrow} & C_{\rightarrow\uparrow} \\ \hline C_{\leftarrow\downarrow} & C_{\rightarrow\downarrow} \end{array}\right)$$

algorithm MATRIXMUL(n, A, B)**:**
  if $n = 2$ **then** **return** $A \times B$ (brute force)
  else
    Partition A into $A_{\leftarrow\uparrow}, A_{\rightarrow\uparrow}, A_{\leftarrow\downarrow}, A_{\rightarrow\downarrow}$
    Partition B into $B_{\leftarrow\uparrow}, B_{\rightarrow\uparrow}, B_{\leftarrow\downarrow}, B_{\rightarrow\downarrow}$
    $C_{\leftarrow\uparrow} \leftarrow$ MATRIXMUL($n/2, A_{\leftarrow\uparrow}, B_{\leftarrow\uparrow}$)+MATRIXMUL($n/2, A_{\rightarrow\uparrow}, B_{\leftarrow\downarrow}$)
    $C_{\rightarrow\uparrow} \leftarrow$ MATRIXMUL($n/2, A_{\leftarrow\uparrow}, B_{\rightarrow\uparrow}$)+MATRIXMUL($n/2, A_{\rightarrow\uparrow}, B_{\rightarrow\downarrow}$)
    $C_{\leftarrow\downarrow} \leftarrow$ MATRIXMUL($n/2, A_{\leftarrow\downarrow}, B_{\leftarrow\uparrow}$)+MATRIXMUL($n/2, A_{\rightarrow\downarrow}, B_{\rightarrow\downarrow}$)
    $C_{\rightarrow\downarrow} \leftarrow$ MATRIXMUL($n/2, A_{\leftarrow\downarrow}, B_{\rightarrow\uparrow}$)+MATRIXMUL($n/2, A_{\rightarrow\downarrow}, B_{\rightarrow\downarrow}$)
    Combine $C_{\leftarrow\uparrow}, C_{\rightarrow\uparrow}, C_{\leftarrow\downarrow}, C_{\rightarrow\downarrow}$ into C
    return C

- $T(n) = 8T(n/2) + n^2$, $T(2) = 8 \Rightarrow T(n) = O(n^3)$ (bummer!)

Fast matrix multiplication (cont'd)

- To improve complexity we try to compute $C_{\leftarrow\uparrow}, C_{\rightarrow\uparrow}, C_{\leftarrow\downarrow}, C_{\rightarrow\downarrow}$ using less than 8 matrix multiplication operations
- Strassen's definitions:

$$
\begin{aligned}
P &= (A_{\leftarrow\uparrow} + A_{\rightarrow\uparrow})(B_{\leftarrow\uparrow} + B_{\rightarrow\downarrow})\\
Q &= (A_{\rightarrow\uparrow} + A_{\rightarrow\downarrow})B_{\leftarrow\uparrow}\\
R &= A_{\leftarrow\uparrow}(B_{\rightarrow\uparrow} - B_{\rightarrow\downarrow})\\
S &= A_{\rightarrow\downarrow}(B_{\rightarrow\uparrow} - B_{\leftarrow\uparrow})\\
T &= (A_{\leftarrow\uparrow} + A_{\rightarrow\uparrow})B_{\rightarrow\downarrow}\\
U &= (A_{\rightarrow\uparrow} - A_{\leftarrow\uparrow})(B_{\leftarrow\uparrow} + B_{\rightarrow\uparrow})\\
V &= (A_{\rightarrow\uparrow} - A_{\rightarrow\downarrow})(B_{\rightarrow\uparrow} + B_{\rightarrow\downarrow})
\end{aligned}
$$

so

$$
\begin{aligned}
C_{\leftarrow\uparrow} &= P + S - T + V\\
C_{\rightarrow\uparrow} &= R + T\\
C_{\rightarrow\uparrow} &= Q + S\\
C_{\rightarrow\downarrow} &= P + R - Q + U
\end{aligned}
$$

- Only 7 multiplication operations!
- $T(n) = 7T(n/2) + n^2, T(2) = 8 \Rightarrow T(n) = O(n^{log 7}) = O(n^{2.81})$
  - Subsequent algorithms were able to bring complexity down to $O(n^{2.373})$

Fast matrix multiplication (cont'd)

- To improve complexity we try to compute $C_{\leftarrow\uparrow}, C_{\rightarrow\uparrow}, C_{\leftarrow\downarrow}, C_{\rightarrow\downarrow}$ using less than 8 matrix multiplication operations
- Strassen's definitions:

$$
\begin{aligned}
P &= (A_{\leftarrow\uparrow} + A_{\rightarrow\uparrow})(B_{\leftarrow\uparrow} + B_{\rightarrow\downarrow}) \\
Q &= (A_{\rightarrow\uparrow} + A_{\rightarrow\downarrow})B_{\leftarrow\uparrow} \\
R &= A_{\leftarrow\uparrow}(B_{\rightarrow\uparrow} - B_{\rightarrow\downarrow}) \\
S &= A_{\rightarrow\downarrow}(B_{\rightarrow\uparrow} - B_{\leftarrow\uparrow}) \\
T &= (A_{\leftarrow\uparrow} + A_{\rightarrow\uparrow})B_{\rightarrow\downarrow} \\
U &= (A_{\rightarrow\uparrow} - A_{\leftarrow\uparrow})(B_{\leftarrow\uparrow} + B_{\rightarrow\uparrow}) \\
V &= (A_{\rightarrow\uparrow} - A_{\rightarrow\downarrow})(B_{\rightarrow\uparrow} + B_{\rightarrow\downarrow})
\end{aligned}
\qquad \text{so} \qquad
\begin{aligned}
C_{\leftarrow\uparrow} &= P + S - T + V \\
C_{\rightarrow\uparrow} &= R + T \\
C_{\rightarrow\uparrow} &= Q + S \\
C_{\rightarrow\downarrow} &= P + R - Q + U
\end{aligned}
$$

- Only 7 multiplication operations!
- $T(n) = 7T(n/2) + n^2, T(2) = 8 \Rightarrow T(n) = O(n^{log7}) = O(n^{2.81})$
  - Subsequent algorithms were able to bring complexity down to $O(n^{2.373})$
- Trick used: split into fewer (but less obvious) sub-problems

Large integer multiplication

Manipulate big integers → represented by arrays of n digits

- Obvious lower bound for addition and multiplication: $\Omega(n)$
- The straightforward algorithms are optimal for addition ($O(n)$) but not necessarily for multiplication ($O(n^2)$)

LARGE INTEGER MULTIPLICATION

Manipulate big integers → represented by arrays of n digits

- Obvious lower bound for addition and multiplication: $\Omega(n)$
- The straightforward algorithms are optimal for addition ($O(n)$) but not necessarily for multiplication ($O(n^2)$)
- Divide and conquer approach:
  - Let u and v be two n-digit integers
  - Let $m = n/2$ and let $u = x \times 10^m + y$ and $v = w \times 10^m + z$
  - It follows that $u \times v = (x \times 10^m + y)(w \times 10^m + z) = xw \times 10^{2m} + (xz + yw) \times 10^m + yz$

```
algorithm IntMul(n, u, v):
    m ← n/2
    if u = 0 ∨ v = 0 then return 0
    else if n = 2 then return u × v
    else
        x ← u DIV 10^m     // most significant m digits
        y ← u REM 10^m     // least significant m digits
        w ← v DIV 10^m
        z ← v REM 10^m
        return  IntMul(m, x, w) × 10^2m
                +(IntMul(m, x, z)
                  +IntMul(m, y, w)) × 10^m
                +IntMul(m, y, z)
```

LARGE INTEGER MULTIPLICATION

Manipulate big integers $\rightarrow$ represented by arrays of n digits

- Obvious lower bound for addition and multiplication: $\Omega(n)$
- The straightforward algorithms are optimal for addition ($O(n)$) but not necessarily for multiplication ($O(n^2)$)
- Divide and conquer approach:
  - Let u and v be two n-digit integers
  - Let $m = n/2$ and let $u = x \times 10^m + y$ and $v = w \times 10^m + z$
  - It follows that
    $u \times v = (x \times 10^m + y)(w \times 10^m + z) = xw \times 10^{2m} + (xz + yw) \times 10^m + yz$

```
algorithm IntMul(n, u, v):
    m ← n/2
    if u = 0 ∨ v = 0 then return 0
    else if n = 2 then return u × v
    else
        x ← u DIV 10^m        // most significant m digits
        y ← u REM 10^m        // least significant m digits
        w ← v DIV 10^m
        z ← v REM 10^m
        return IntMul(m, x, w) × 10^(2m)
               +(IntMul(m, x, z)
                 +IntMul(m, y, w)) × 10^m
               +IntMul(m, y, z)
```

- Running time:
  $T(n) = 4T(n/2) + n$,
  $T(2) = 4$
- Complexity: $O(n^2)$

- Improvement:
  - Let $p_1 = xw$, $p_2 = yz$, $p_3 = (x+y)(w+z)$
  - Then $p_3 - p_1 - p_2 = (x+y)(w+z) – xw – yz = xz + yw$
  - Then $p = (x \times 10^m + y)(w \times 10^m + z) =$ $xw \times 10^{2m} + (xz + yw) \times 10^m + yz = p_1 10^{2m} + (p3 - p1 - p2)10^m + p_2$

algorithm FASTMUL(n, u, v)**:**

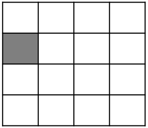
$m \leftarrow n/2$
if $u = 0 \vee v = 0$ **then return** 0
else if $n = 2$ **then**
  return $u \times v$
else
  $x \leftarrow u$ DIV 10^m
  $y \leftarrow u$ REM 10^m
  $w \leftarrow v$ DIV 10^m
  $z \leftarrow v$ REM 10^m
  $p_1 =$ FASTMUL(m, x, w)
  $p_2 =$ FASTMUL(m, y, z)
  $p_3 =$ FASTMUL$(m, x + y, w + z)$
  return $p_1 10^{2m} + (p3 - p1 - p2)10^m + p^2$

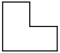
- Running time: $T(n) = 3T(n/2) + n$, $T(2) = 4$
- Complexity: $O(n^{\log 3}) = O(n^{1.585})$

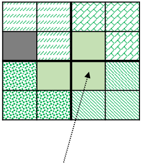
TROMINO TILING

Tile a bathroom floor ("board") with trominos without covering the drain (designated square on the board)

algorithm TILE(B, n, L): // B is the $n \times n$ board, L is the drain location
- **if** $n = 2$ **then**
  - Tile with one tromino without covering L
- **else**
  - Divide B into 4 $n/2 \times n/2$ sub-boards $B_1, \ldots, B_4$
  - Place a tromino to cover one square on each board that does not contain L
  - Let $L_1, \ldots L_4$ be the squares on each sub-board that are either covered or L
  - **for** $i = 1$ **to** 4 **do**
    - TILE(B_i,$n/2$,L_i)



Tromino 



Running time/trominoes used:

- $T(n) = 4T(n/2) + 1,\ T(2) = 1$
- $T(n) = 1/3(n^2 - 1)$
- Much better than the trial and error approach

When NOT to use divide and conquer

- Divide and conquer does not work for everything
- The crux of the technique is the ability to divide a problem into-sub problems
- Therefore divide and conquer is not the right thing to do when:
  - The size of sub-problems is the same (or larger) than the size of the original problem
    - Example: initial version of matrix or integer multiplication
    - Dramatic example: computing Fibonacci numbers
  - When the process of splitting into sub-problems takes too much time
  - When the process of combining the sub-solutions takes too much time