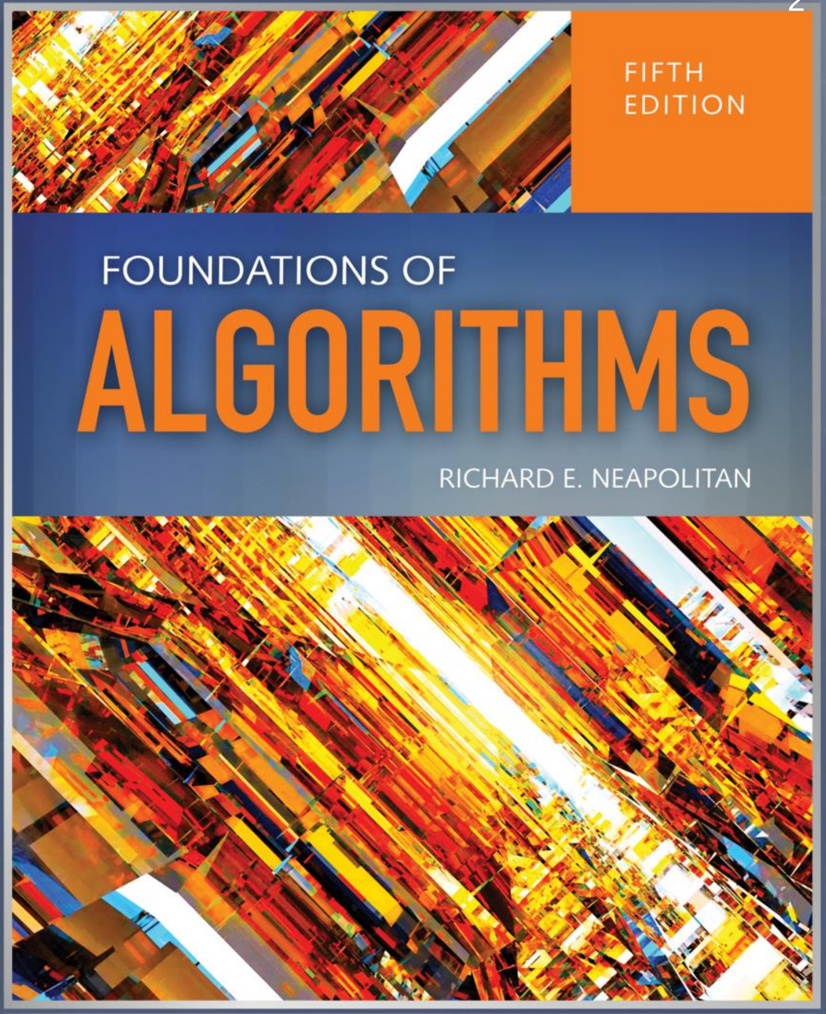
Basic algorithmic techniques

When faced with a new algorithmic problem, one should consider applying one of the following approaches:

- **Divide-and-conquer** :: divide the problem into two subproblems, solve each problem separately and merge the solutions
- **Dynamic programming** :: express the solution of the original problem as a recursion on solutions of similar smaller problems. Then instead of solving only the original problem, solve all sub-problems that can occur when the recursion is unravelled, and combine their solutions
- **Greedy approach** :: build the solution of an optimization problem one piece at a time, optimizing each piece separately
- **Inductive approach** :: express the solution of the original problem based on the solution of the same problem with one fewer item; a special case of dynamic programming and similar to the greedy approach



Divide and Conquer

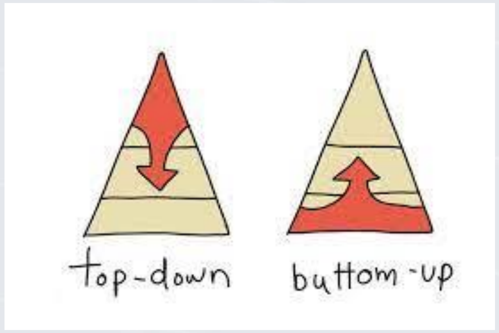
Chapter 2

Objectives

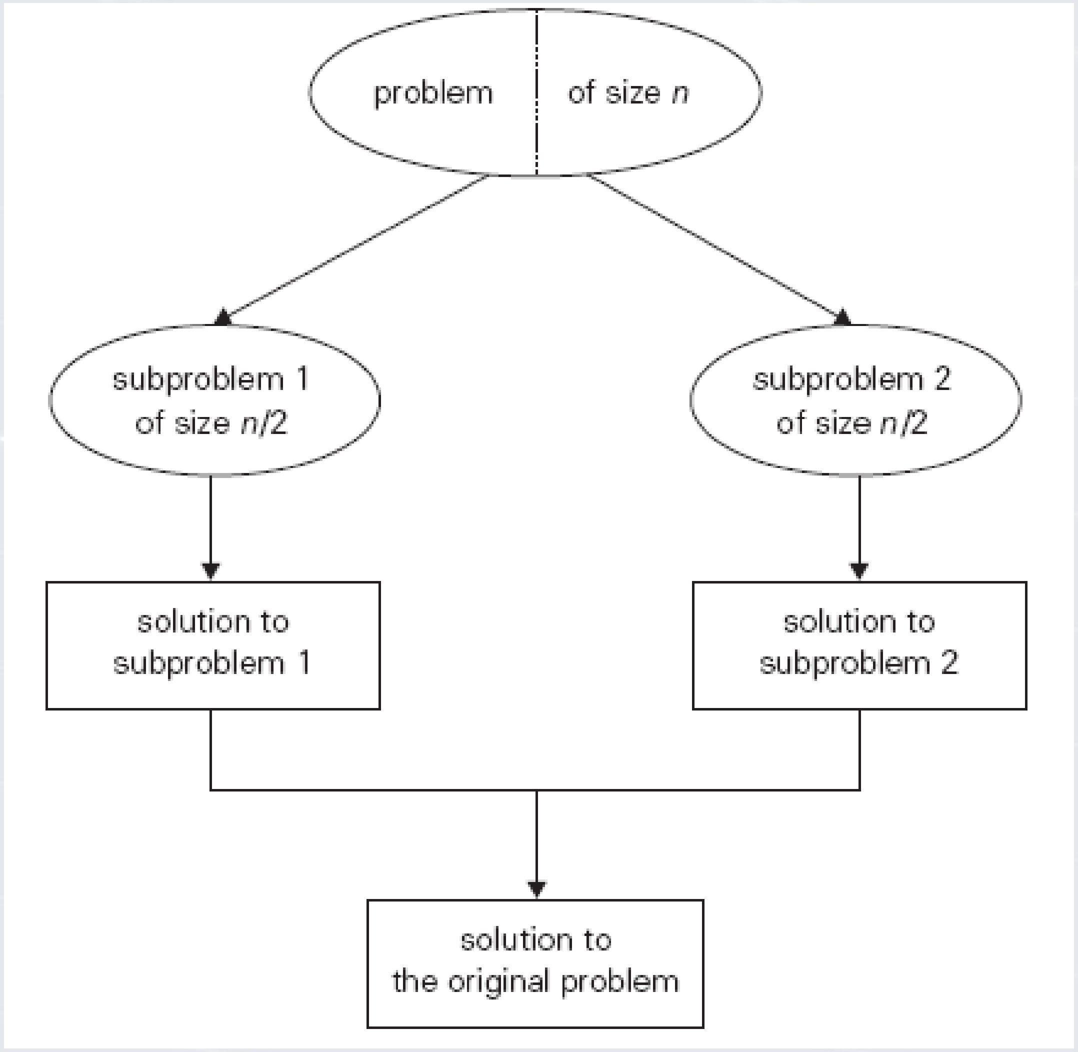
- Describe the **divide-and-conquer approach** to solving problems
- Apply the divide-and-conquer approach to solve a problem
- Determine when the divide-and-conquer approach is an appropriate solution approach
- Determine complexity analysis of divide and conquer algorithms
- Contrast worst-case and average-case complexity analysis
- **Master Method** to analyze recursive algorithms

Battle of Austerlitz
December 2, 1805

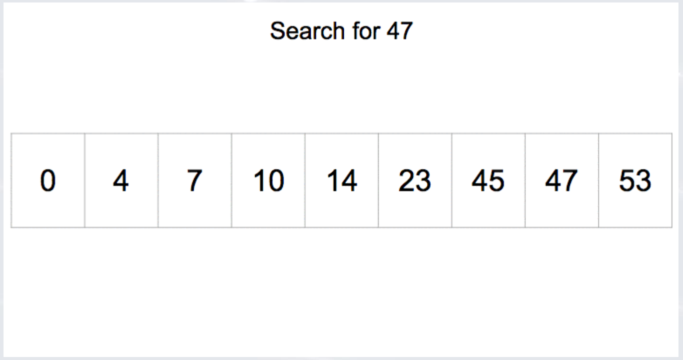
- Napoleon split the Austro-Russian Army and was able to conquer 2 weaker armies
- Divide an instance of a problem into 2 or more smaller instances
- Top-down approach



Divide-and-Conquer Technique



Binary Search



- Locate key x in an array of size n sorted in non-decreasing order
- Compare x with the middle element – if equal, done – quit. Else
  - *Divide* the Array into two sub-arrays approximately half as large
    - If x is smaller than the middle item, select left sub-array
    - If x is larger than the middle item, select right sub-array

Binary Search

- *Conquer* (solve) the sub-array: Is x in the sub-array using recursion until the sub-array is sufficiently small?
- *Obtain* the solution to the array from the solution to the sub-array

Binary Search (Recursive)

Problem: Determine whether x is in the sorted array S of size n.

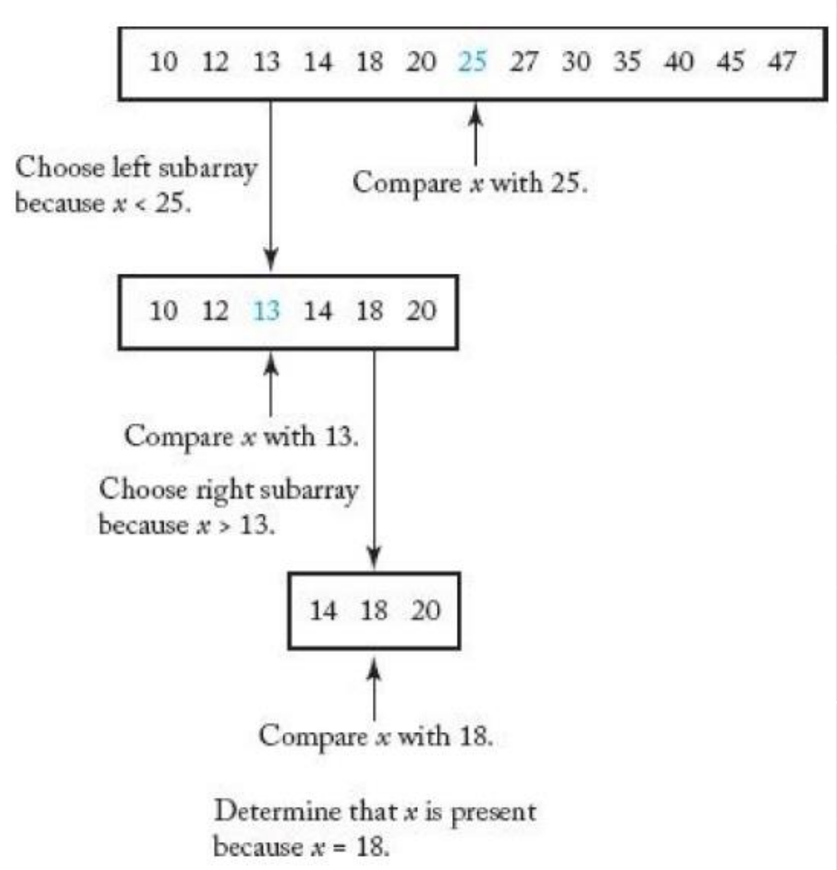
Inputs: positive integer n, sorted (nondecreasing order) array of keys S indexed from 1 to n, a key x.

Outputs: *location*, the location of x in S (0 if x is not in S).

$x = 18$

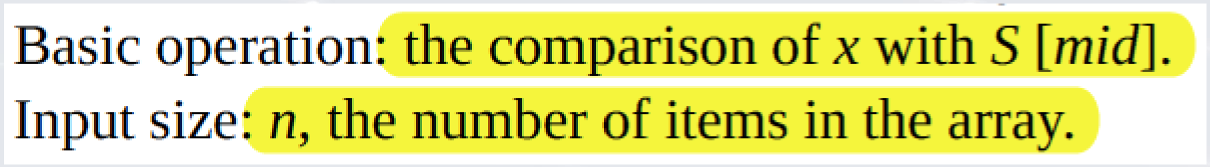
```
index location (index low, index high)
{
  index mid;

  if (low > high)
    return 0;
  else {
    mid = ⌊(low + high)/2⌋;
    if (x == S[mid])
      return mid
    else if (x < S[mid])
      return location(low, mid − 1);
    else
      return location(mid + 1, high);
  }
}
```



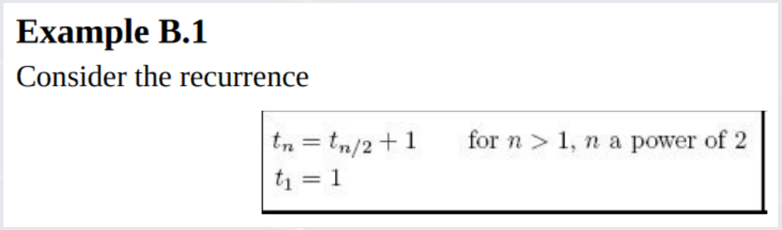
locationout = *location* (1 , *n*) ;

Worst Case Complexity Analysis


Basic operation: the comparison of *x* with *S* [*mid*].

Input size: *n*, the number of items in the array.

- n is a power of 2 and x > S[n]
- W(n) = W(n/2) + 1 for n>1 and n power of 2
- W(1) = 1
  - W(n/2) = the number of comparisons in the recursive call
  - 1 comparison at the top level
- Example B1 in Appendix B:
  - W(n) = lg n + 1
- n not a power of 2
  - W(n) = ⌊lg n ⌋ +1 ϵ θ(lg n)


Example B.1

Consider the recurrence

$$\begin{aligned} t_n &= t_{n/2} + 1 && \text{for } n > 1,\ n \text{ a power of } 2 \\ t_1 &= 1 \end{aligned}$$

Merge sort [CLRS 2.3.1]

Merge sort is a divide-and-conquer algorithm.

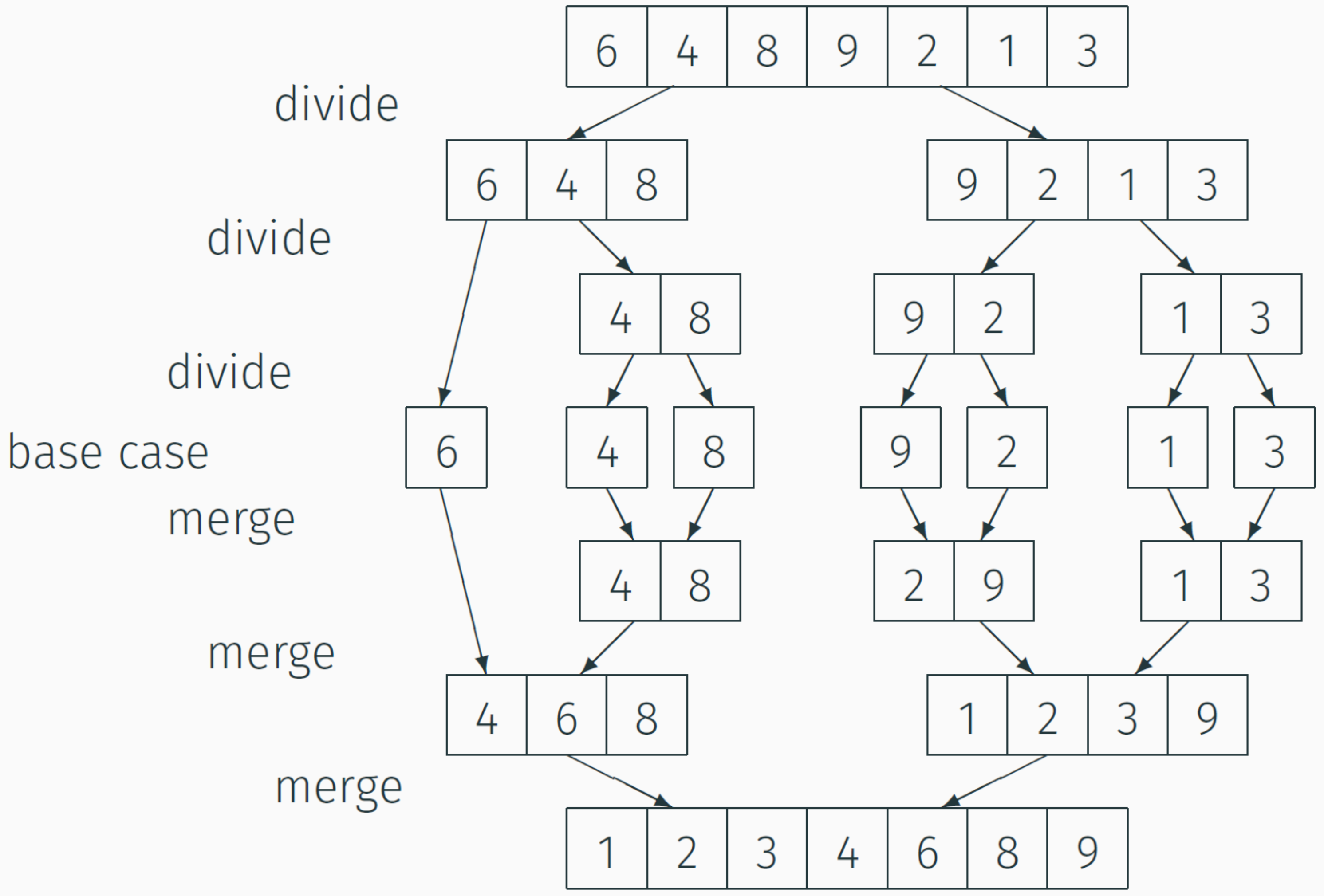
Informal description:

It sorts a subarray $A[p\,..\,r) := A[p\,..\,r-1]$

Divide	by splitting it into subarrays $A[p\,..\,q)$ and $A[q\,..\,r)$ where $q = \lfloor (p+r)/2 \rfloor$.
Conquer	by recursively sorting the subarrays. Recursion stops when the subarray contains only one element.
Combine	by merging the *sorted* subarrays $A[p\,..\,q)$ and $A[q\,..\,r)$ into a single sorted array, using a procedure called $\textsc{Merge}(A, p, q, r)$.

$\textsc{Merge}$ compares the two smallest elements of the two subarrays and copies the smaller one into the output array.
This procedure is repeated until all the elements in the two subarrays have been copied.

Example



6 4 8 9 2 1 3
divide
6 4 8
9 2 1 3
divide
4 8
9 2
1 3
divide
base case
6
4
8
9
2
1
3
merge
4 8
2 9
1 3
merge
4 6 8
1 2 3 9
merge
1 2 3 4 6 8 9

Pseudocode for MERGE-SORT

```
MERGE-SORT$(A, p, r)$
    Input: An integer array $A$ with indices $p < r$.
    Output: The subarray $A[p \mathinner{.\,.} r)$ is sorted in non-decreasing order.
1   if $r > p + 1$
2       $q = \lfloor (p + r)/2 \rfloor$
3       MERGE-SORT$(A, p, q)$
4       MERGE-SORT$(A, q, r)$
5       MERGE$(A, p, q, r)$
```

Initial call: MERGE-SORT$(A, 1, n + 1)$

Merge

Input: Array A with indices p, q, r such that

- $p < q < r$
- Subarrays $A[p \,..\, q)$ and $A[q \,..\, r)$ are both sorted.

Output: The two sorted subarrays are merged into a single sorted subarray in $A[p \,..\, r)$.

Pseudocode for Merge

```
Merge($A, p, q, r$)
$n_1 = q - p$
$n_2 = r - q$
Create array $L$ of size $n_1 + 1$
Create array $R$ of size $n_2 + 1$
for $i = 1$ to $n_1$
    $L[i] = A[p + i - 1]$
for $j = 1$ to $n_2$
    $R[j] = A[q + j - 1]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ to $r - 1$
    if $L[i] \leq R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$
```

Running time of MERGE

- The first two **for** loops take $\Theta(n_1 + n_2) = \Theta(n)$ time, where $n = r - p$.
- The last **for** loop makes n iterations, each taking constant time, for $\Theta(n)$ time.
- Total time: $\Theta(n)$.

Remark

- The test in line 14 is left-biased, which ensures that MERGE-SORT is a *stable* sorting algorithm: if $A[i] = A[j]$ and $A[i]$ appears before $A[j]$ in the input array, then in the output array the element pointing to $A[i]$ appears to the left of the element pointing to $A[j]$.

Characteristics of merge sort

- The worst-case running time of MERGE-SORT is $\Theta(n \log n)$, much better that the worst-case running time of INSERTION-SORT, which was $\Theta(n^2)$.
  (see next slides for the explicit analysis of MERGE-SORT).
- MERGE-SORT is stable, because MERGE is left-biased.
- MERGE and therefore MERGE-SORT is not in-place:
  it requires $\Theta(n)$ extra space.
- MERGE-SORT is not an online-algorithm: the whole array A must be specified before the algorithm starts running.

Analysing divide-and-conquer algorithms [CLRS 2.3.2]

We often use a *recurrence* to express the running time of a divide-and-conquer algorithm.

Let $T(n)$ = running time on a problem of size n.

- If n is small (say $n \leq \ell$), use constant-time brute force solution.
- Otherwise, we divide the problem into a subproblems, each $1/b$ the size of the original.
- Let the time to divide a size-n problem be $D(n)$.
- Let the time to combine solutions (back to that of size n) be $C(n)$.

We get the recurrence

$$T(n) = \begin{cases} c & \text{if } n \leq \ell \\ a\,T(n/b) + D(n) + C(n) & \text{if } n > \ell \end{cases}$$

Example: MERGE-SORT

For simplicity, assume $n = 2^k$.

For $n = 1$, the running time is a constant c.

For $n \geq 2$, the time taken for each step is:

- **Divide**: Compute $q = (p + r)/2$; so, $D(n) = \Theta(1)$.
- **Conquer**: Recursively solve 2 subproblems, each of size $n/2$; so, $2T(n/2)$.
- **Combine**: MERGE two arrays of size n; so, $C(n) = \Theta(n)$.

More precisely, the recurrence for MERGE-SORT is

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2\,T(n/2) + f(n) & \text{if } n > 1 \end{cases}$$

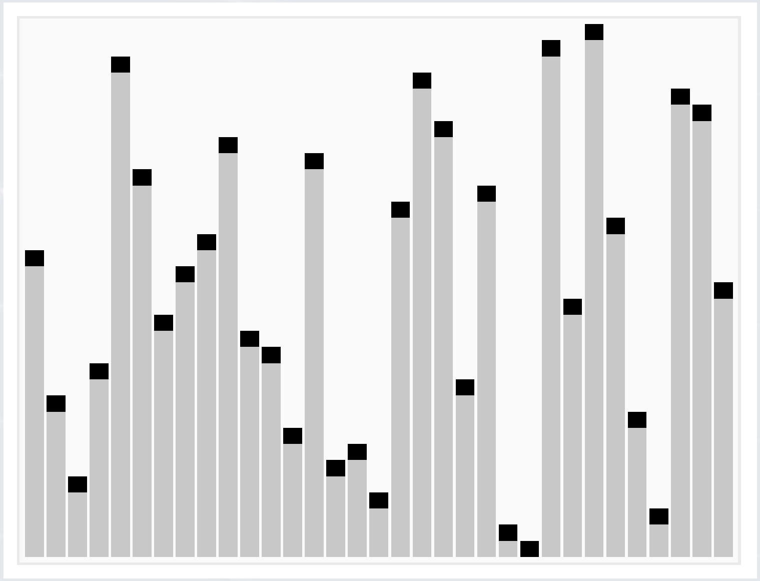
where the function $f(n)$ is bounded as $d'\,n \leq f(n) \leq d\,n$ for suitable constants $d, d' > 0$.

Worst-Case Analysis Mergesort

https://www.youtube.com/watch?v=dENca26N6V4

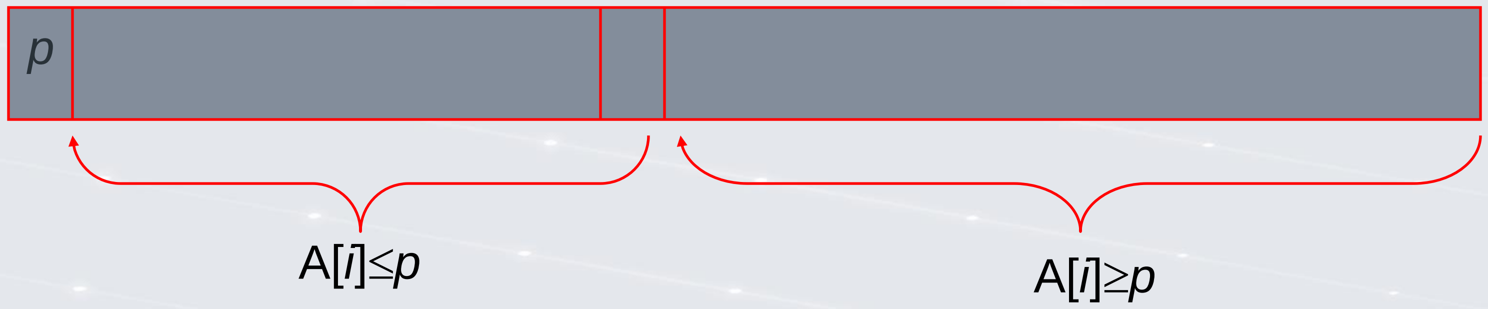
QuickSort

- Normally considered the best general purpose sort algorithm
- Implemented in many standard libraries
  - Array.Sort in C# /.net
  - Array.Sort in Java
  - qsort (C standard library)*



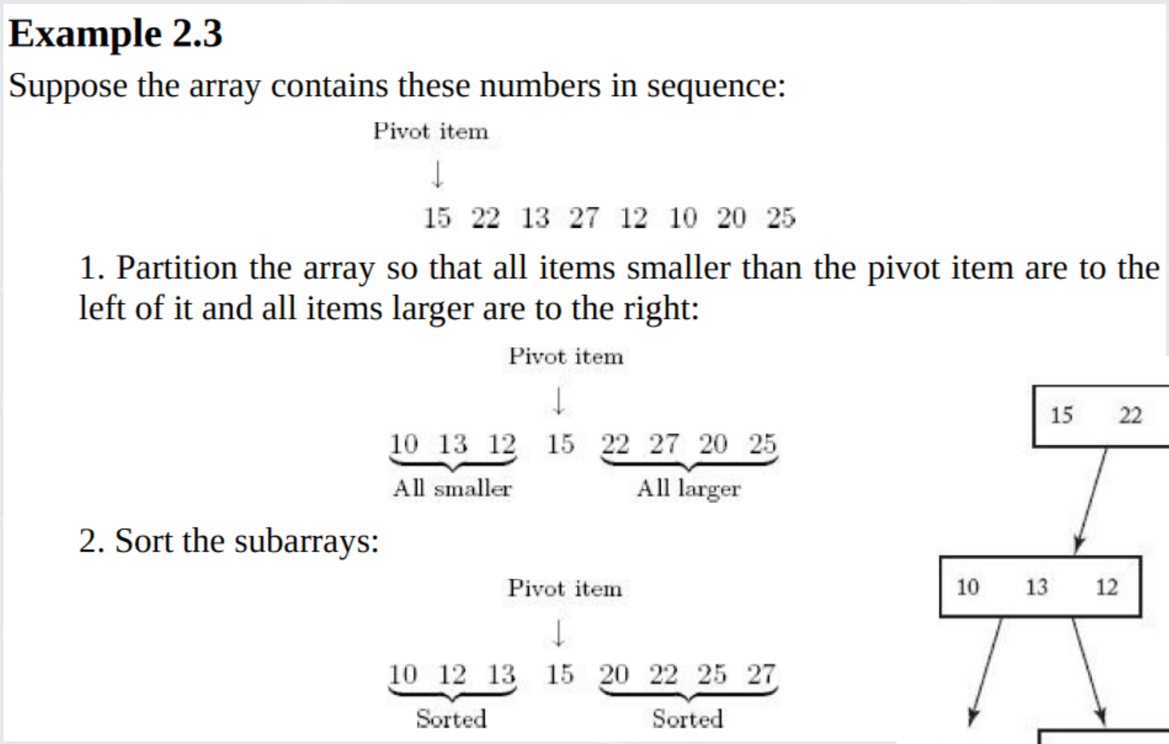
Quicksort

- Select a *pivot* (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first *s* positions are smaller than or equal to the pivot and all the elements in the remaining *n*-*s* positions are larger than or equal to the pivot



- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

QuickSort



Example 2.3

Suppose the array contains these numbers in sequence:

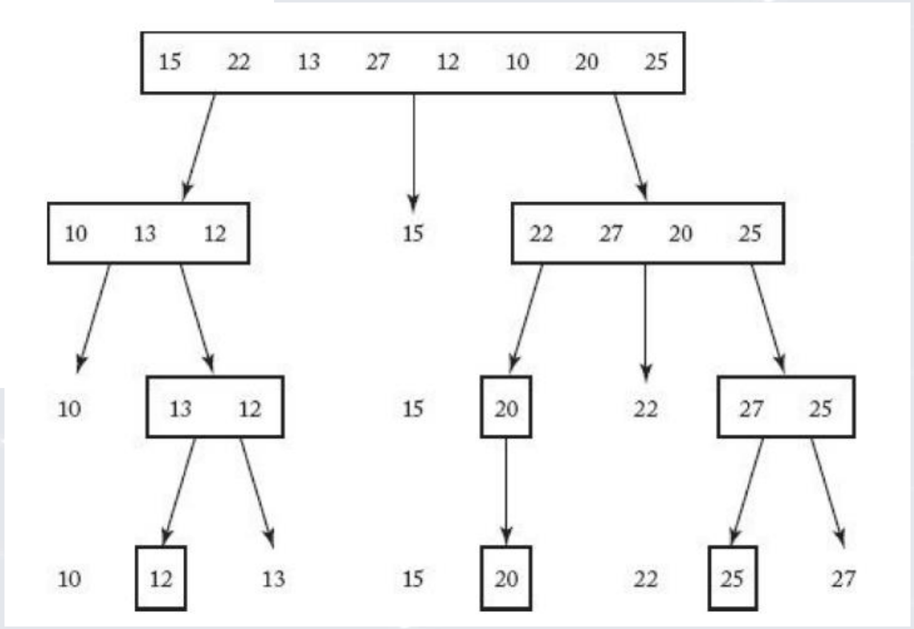
$$\begin{array}{c} \text{Pivot item} \\ \downarrow \\ 15 \quad 22 \quad 13 \quad 27 \quad 12 \quad 10 \quad 20 \quad 25 \end{array}$$

1. Partition the array so that all items smaller than the pivot item are to the left of it and all items larger are to the right:

$$\begin{array}{c} \text{Pivot item} \\ \downarrow \\ \underbrace{10 \quad 13 \quad 12}_{\text{All smaller}} \quad 15 \quad \underbrace{22 \quad 27 \quad 20 \quad 25}_{\text{All larger}} \end{array}$$

2. Sort the subarrays:

$$\begin{array}{c} \text{Pivot item} \\ \downarrow \\ \underbrace{10 \quad 12 \quad 13}_{\text{Sorted}} \quad 15 \quad \underbrace{20 \quad 22 \quad 25 \quad 27}_{\text{Sorted}} \end{array}$$



Quicksort

Problem: Sort *n* keys in nondecreasing order.

Inputs: positive integer *n*, array of keys *S* indexed from 1 to *n*.

Outputs: the array *S* containing the keys in nondecreasing order.

quicksort (1 , *n*) ;

```
void quicksort (index low, index high)
{
  index pivotpoint;

  if (high > low){
     partition(low, high, pivotpoint);
     quicksort(low, pivotpoint − 1);
     quicksort(pivotpoint + 1, high);
  }
}
```

Partition

Problem: Partition the array *S* for Quicksort.

Inputs: two indices, *low* and *high*, and the subarray of *S* indexed from *low* to *high*.

Outputs: *pivotpoint*, the pivot point for the subarray indexed from *low* to *high*.

```
void partition (index low, index high,
                index& pivotpoint)
{
   index i, j;
   keytype pivotitem;

   pivotitem = S[low];
   j = low;
   for (i = low + 1; i <= high; i++)
      if (S[i] < pivotitem){
         j++;
         exchange S[i] and S[j];
      }
   pivotpoint = j;
   exchange S[low] and S[pivotpoint];
}
```

Basic operation

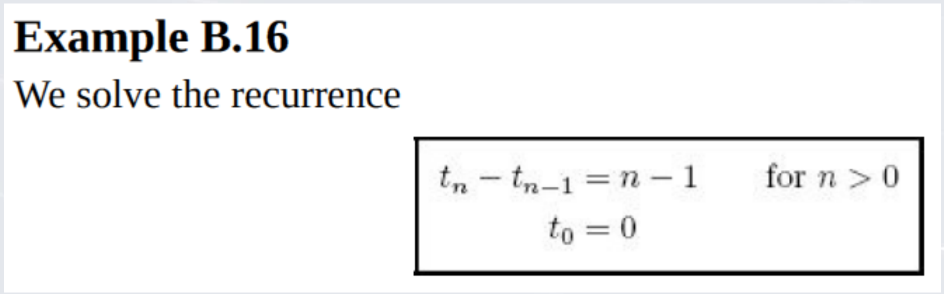
- Comparison of S[i] with pivot item
- Input size n

Worst-Case Complexity Analysis of Quicksort

- Array is sorted in non-decreasing order
- Array is repeatedly sorted into an empty sub-array which is less than the pivot and a sub-array of n-1 containing items greater than pivot
- If there are k keys in the current sub-array, k-1 key comparisons are executed

Worst-Case Complexity Analysis of Quicksort

- T(n) is specified because analysis is for the every-case complexity for the class of instances already sorted in non-decreasing order
- T(n) = time to sort left sub-array + time to sort right sub-array + time to partition
- T(n) = T(0) + T(n-1) + n – 1
- T(n) = T(n – 1) + n – 1 for n > 0
- T(0) = 0
- From B16
  - T(n) = n(n-1)/2



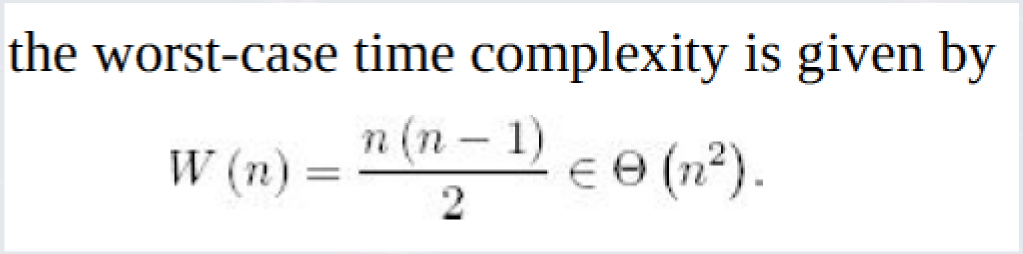
Example B.16

We solve the recurrence

$$\boxed{\begin{aligned} t_n - t_{n-1} &= n - 1 \qquad \text{for } n > 0 \\ t_0 &= 0 \end{aligned}}$$

Worst Case

- At least n(n-1)/2
- Use induction to show it is the worst case
  - W(n)<=n(n-1)/2



the worst-case time complexity is given by

$$W(n) = \frac{n(n-1)}{2} \in \Theta(n^2).$$

Average-Case Time Complexity of Quicksort

- Value of the pivot point is equally likely to be any of the numbers from 1 to n
- Average obtained is the average sorting time when every possible ordering is sorted the same number of times
- *An= p=1n1n(Ap-1+ An-p+n-1)*

$$A(n) = \sum_{p=1}^{n} \frac{1}{n}(A(p-1) + A(n-p)) + n - 1$$

- 1/n is the probability pivot point is p

- Solving equation and using B22
  - **A(n) ϵ θ(n*lgn)**

Analysis of Quicksort

- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array! — $\Theta(n^2)$
- Average case: random arrays — $\Theta(n \log n)$

- Improvements:
  - better pivot selection: median of three partitioning
  - switch to insertion sort on small subfiles
  - elimination of recursion

  These combine to 20-25% improvement

- Considered the method of choice for internal sorting of large files ($n \geq$ 10000)

Multiplication of Large Integers

Consider the problem of multiplying two (large) n-digit integers represented by arrays of their digits such as:

A = 12345678901357986429   B = 87654321284820912836

The grade-school algorithm:

$$\begin{array}{r} a_1\ a_2 \ldots\ a_n \\ \underline{b_1\ b_2 \ldots\ b_n} \\ (d_{10})\ d_{11}d_{12} \ldots d_{1n} \\ (d_{20})\ d_{21}d_{22} \ldots d_{2n} \qquad\quad \\ \ldots \ldots \ldots \ldots \ldots \ldots \ldots \qquad\quad \\ \underline{(d_{n0})\ d_{n1}d_{n2} \ldots d_{nn} \qquad\qquad\qquad} \end{array}$$

Efficiency: n^2 one-digit multiplications

Multiplying Two-Digit Integers:
74 × 43
Step 1: Find 74×3
Step 2: Find 74×40
Step 3: Add!
74 × 43 = 222
74 × 43 = 222, 0
74 × 43 = 222, 2960
74 × 43 = 222 + 2960 = 3182
(drop down a 0)
(find 74×4)

Multiplications of Large Integers

$$\underbrace{567,832}_{6\text{ digits}} = \underbrace{567}_{3\text{ digits}} \times 10^3 + \underbrace{832}_{3\text{ digits}}$$

$$\underbrace{9,423,723}_{7\text{ digits}} = \underbrace{9423}_{4\text{ digits}} \times 10^3 + \underbrace{723}_{3\text{ digits}}$$

In general, if *n* is the number of digits in the integer *u*, we will split the integer into two integers, one with $\lceil n/2 \rceil$ and the other with $\lfloor n/2 \rfloor$, as follows:

$$\underbrace{u}_{n\text{ digits}} = \underbrace{x}_{\lceil n/2 \rceil\text{ digits}} \times 10^m + \underbrace{y}_{\lfloor n/2 \rfloor\text{ digits}}$$

$$567,832 \times 9,423,723 =$$

$$\begin{aligned} &= (567 \times 10^3 + 832)\,(9423 \times 10^3 + 723) \\ &= 567 \times 9423 \times 10^6 + (567 \times 723 + 9423 \times 832) \\ &\quad \times 10^3 + 832 \times 723 \end{aligned}$$

With this representation, the exponent *m* of 10 is given by

$$m = \left\lfloor \frac{n}{2} \right\rfloor.$$

If we have two *n*-digit integers

$$\begin{aligned} u &= x \times 10^m + y \\ v &= w \times 10^m + z, \end{aligned}$$

their product is given by

$$\begin{aligned} uv &= (x \times 10^m + y)\,(w \times 10^m + z) \\ &= xw \times 10^{2m} + (xz + wy) \times 10^m + yz. \end{aligned}$$

First Divide-and-Conquer Algorithm

A small example: $A * B$ where $A = 2135$ and $B = 4014$

$A = (21 \cdot 10^2 + 35)$, $B = (40 \cdot 10^2 + 14)$

So, $A * B = (21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14)$

$= 21 * 40 \cdot 10^4 + (21 * 14 + 35 * 40) \cdot 10^2 + 35 * 14$

In general, if $A = A_1A_2$ and $B = B_1B_2$ (where A and B are *n*-digit, A_1, A_2, B_1, B_2 are *n*/2-digit numbers),

$$\mathbf{A * B = A_1 * B_1 \cdot 10^{n} + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2}$$

Recurrence for the number of one-digit multiplications M(*n*):

$$M(n) = 4M(n/2), \quad M(1) = 1$$

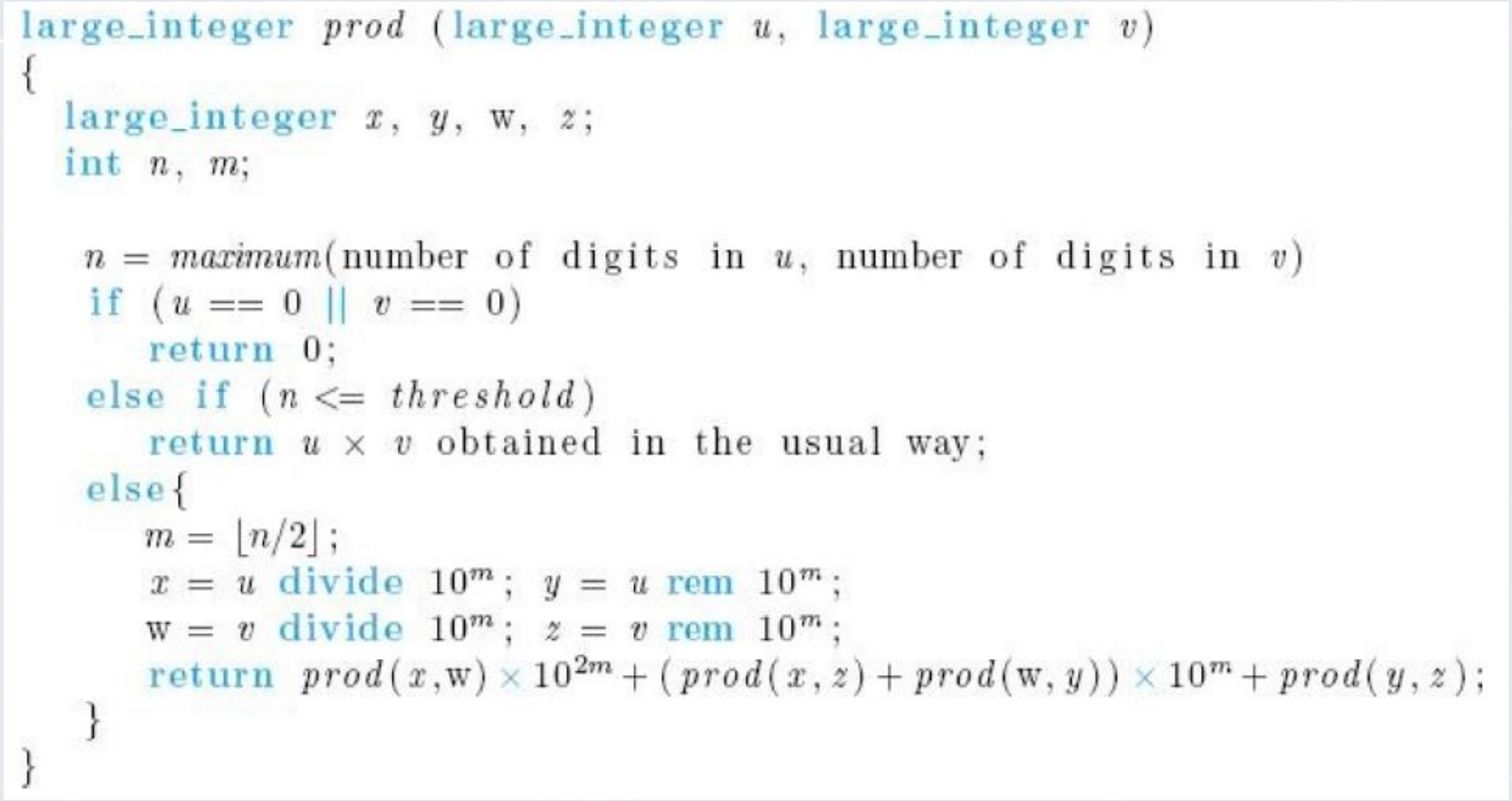
Solution: $M(n) = n^2$

Large Integer Multiplication

Problem: Multiply two large integers, *u* and *v*.

Inputs: large integers *u* and *v*.

Outputs: *prod*, the product of *u* and *v*.



```
large_integer prod (large_integer $u$, large_integer $v$)
{
  large_integer $x$, $y$, w, $z$;
  int $n$, $m$;

   $n$ = $maximum$(number of digits in $u$, number of digits in $v$)
   if ($u$ == 0 || $v$ == 0)
      return 0;
   else if ($n$ <= $threshold$)
      return $u \times v$ obtained in the usual way;
   else{
      $m = \lfloor n/2 \rfloor$;
      $x$ = $u$ divide $10^m$; $y$ = $u$ rem $10^m$;
      w = $v$ divide $10^m$; $z$ = $v$ rem $10^m$;
      return $prod(x,w) \times 10^{2m} + (prod(x,z) + prod(w,y)) \times 10^m + prod(y,z)$;
   }
}
```

Second Divide-and-Conquer Algorithm

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

The idea is to decrease the number of multiplications from 4 to 3:

$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) + A_2 * B_2,$

I.e., $(A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2,$
which requires only 3 multiplications at the expense of (4-1) extra add/sub.

Recurrence for the number of multiplications M(n):

$$M(n) = 3M(n/2), \quad M(1) = 1$$

Solution: $M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$

Large Integer Multiplication 2

Problem: Multiply two large integers, *u* and *v*.

Inputs: large integers *u* and *v*.

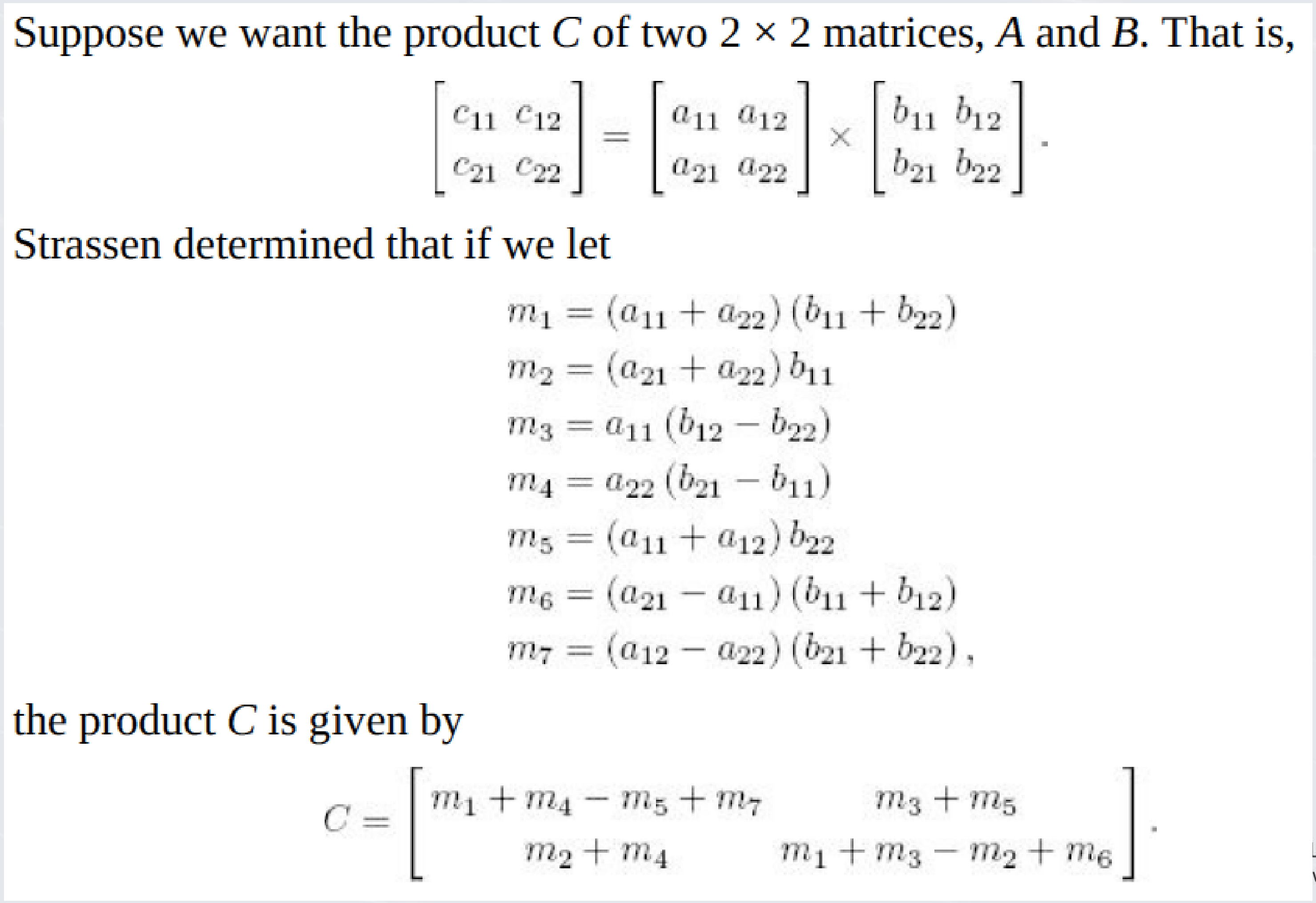
Outputs: *prod2*, the product of *u* and *v*.

```
large_integer prod2 (large_integer u, large_integer v)
{
  large_integer x, y, w, z, r, p, q;
  int n, m;

  n = maximum(number of digits in u, number of digits in v);
  if ( u == 0 || v == 0)
     return 0;
  else if ( n <= threshold)
     return u × v obtained in the usual way;
  else{
     m = $\lfloor n/2 \rfloor$;
     x = u divide $10^m$; y = u rem $10^m$;
     w = v divide $10^m$; z = v rem $10^m$;
     r = prod2(x + y, w + z);
     p = prod2(x, w);
     q = prod2(y, z);
     return $p \times 10^{2m} + (r - p - q) \times 10^m + q$;
  }
}
```

Strassen’s Matrix Multiplication

Strassen observed [1969] that the product of two matrices can be computed as follows:



Suppose we want the product C of two 2×2 matrices, A and B. That is,

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

Strassen determined that if we let

$$\begin{aligned} m_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ m_2 &= (a_{21} + a_{22})\, b_{11} \\ m_3 &= a_{11}(b_{12} - b_{22}) \\ m_4 &= a_{22}(b_{21} - b_{11}) \\ m_5 &= (a_{11} + a_{12})\, b_{22} \\ m_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\ m_7 &= (a_{12} - a_{22})(b_{21} + b_{22}), \end{aligned}$$

the product C is given by

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}.$$

Strassen's Matrix Multiplication

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix} \qquad B = \begin{bmatrix} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{bmatrix}.$$

$$\overset{\leftarrow n/2 \rightarrow}{} \quad n/2 \left[\begin{array}{c:c} C_{11} & C_{12} \\ \hdashline C_{21} & C_{22} \end{array}\right] = \left[\begin{array}{c:c} A_{11} & A_{12} \\ \hdashline A_{21} & A_{22} \end{array}\right] \times \left[\begin{array}{c:c} B_{11} & B_{12} \\ \hdashline B_{21} & B_{22} \end{array}\right]$$

$$\begin{aligned} M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ &= \left(\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \right) \times \left(\begin{bmatrix} 8 & 9 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix} \right) \\ &= \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix}. \end{aligned}$$

$$\overset{\leftarrow 2 \rightarrow}{} \quad 2 \left[\begin{array}{c:c} C_{11} & C_{12} \\ \hdashline C_{21} & C_{22} \end{array}\right] = \left[\begin{array}{cc:cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hdashline 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{array}\right] \times \left[\begin{array}{cc:cc} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ \hdashline 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{array}\right]$$

$$= \begin{bmatrix} 3 \times 17 + 5 \times 7 & 3 \times 10 + 5 \times 9 \\ 11 \times 17 + 13 \times 7 & 11 \times 10 + 13 \times 9 \end{bmatrix} = \begin{bmatrix} 86 & 75 \\ 278 & 227 \end{bmatrix}.$$

Strassen

Problem: Determine the product of two $n \times n$ matrices where n is a power of 2.

Inputs: an integer n that is a power of 2, and two $n \times n$ matrices A and B.

Outputs: the product C of A and B.

```
void strassen (int n
                   n × n_matrix A,
                   n × n_matrix B,
                   n × n_matrix& C)
{
  if (n <= threshold)
     compute C = A × B using the standard algorithm;
  else{
     partition A into four submatrices A11, A12, A21 ,A22;
     partition B into four submatrices B11, B12, B21 ,B22;
     compute C = A × B using Strassen's method;
        // example recursive call:
        // strassen(n/2, A11 + A22, B11 + B22, M1);
  }
}
```

Analysis of Strassen's Algorithm

If n is not a power of 2, matrices can be padded with zeros.

Number of multiplications:

$$M(n) = 7M(n/2), \quad M(1) = 1$$

Solution: $M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$ vs. n^3 of brute-force alg.

Algorithms with better asymptotic efficiency are known but they are even more complex.

Summary

Recursive approach

Write $x = 10^{n/2}a + b \qquad y = 10^{n/2}c + d$

[where a,b,c,d are n/2 – digit numbers]

So :

$$x \cdot y = 10^n ac + 10^{n/2}(ad + bc) + bd \qquad (*)$$

Algorithm#1 : recursively compute ac,ad,bc,bd, then compute (*) in the obvious way.

T(n) = maximum number of operations this algorithm needs to multiply two n-digit numbers

Recurrence : express T(n) in terms of running time of recursive calls.

Base Case : T(1) <= a constant.

For all n > 1 : $T(n) \le 4T(n/2) + O(n)$

Work done here

Work done by recursive calls

Algorithm #2 (Gauss) : recursively compute ac[(1)], bd[(2)], (a+b)(c+d)[(3)] [recall ad+bc = (3) – (1) – (2)]

New Recurrence :

Base Case : T(1) <= a constant

For all n>1 : $T(n) \le 3T(n/2) + O(n)$

Work done here

Work done by recursive calls

Solving recurrence equations

We will consider three methods for solving recurrence equations:

1. Guess-and-test (called the substitution method in [CLRS])
2. Recursion tree
3. Master Theorem
4. By changing variables

Guess-and-test [CLRS 4.3]

- Guess an expression for the solution. The expression can contain constants that will be determined later.
- Use induction to find the constants and show that the solution works.

Let us apply this method to MERGE-SORT.

The recurrence of MERGE-SORT implies that there exist two constants $c, d > 0$ such that

$$T(n) \leq \begin{cases} c & \text{if } n = 1 \\ 2\,T(n/2) + d\,n & \text{if } n > 1 \end{cases}$$

Guess. There is some constant $a > 0$ such that $T(n) \leq an \lg n$ for all $n \geq 2$ that are powers of 2.

Let's test it!

Solving the MERGE-SORT recurrence by guess-and-test

Test. For $n = 2^k$, by induction on k.

Base case: $k = 1$

$$T(2) = 2c + 2d \leq a\,2\lg 2 \qquad \text{if } a \geq c + d$$

Inductive step: assume $T(n) \leq an \log n$ for $n = 2^k$.
Then, for $n' = 2^{k+1}$ we have:

$$\begin{aligned} T(n') &\leq 2a\tfrac{n'}{2}\lg\left(\tfrac{n'}{2}\right) + d\,n' \\ &= an'\lg n' - an'\lg 2 + d\,n' \\ &\leq an'\lg n' \qquad \text{if } a \geq d \end{aligned}$$

In summary: choosing $a \geq c + d$ ensures $T(n) \leq an\lg n$,
and thus $T(n) = O(n\log n)$.
A similar argument can be used to show that $T(n) = \Omega(n\log n)$.
Hence, $T(n) = \Theta(n\log n)$.

The recursion tree [CLRS 4.4]

Guess-and-test is great, but how do we guess the solution?
One way is to use the *recursion tree*,
which exposes successive unfoldings of the recurrence.

The idea is well exemplified in the case of MERGE-SORT.
The recurrence is

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2\,T(n/2) + f(n) & \text{if } n > 1 \end{cases}$$

where the function $f(n)$ satisfies the bounds $d'\,n \leq f(n) \leq d\,n$, for suitable constants $d, d' > 0$.

Unfolding the recurrence of MERGE-SORT

Assume $n = 2^k$ for simplicity.

First unfolding: cost of $f(n)$ plus cost of two subproblems of size $n/2$

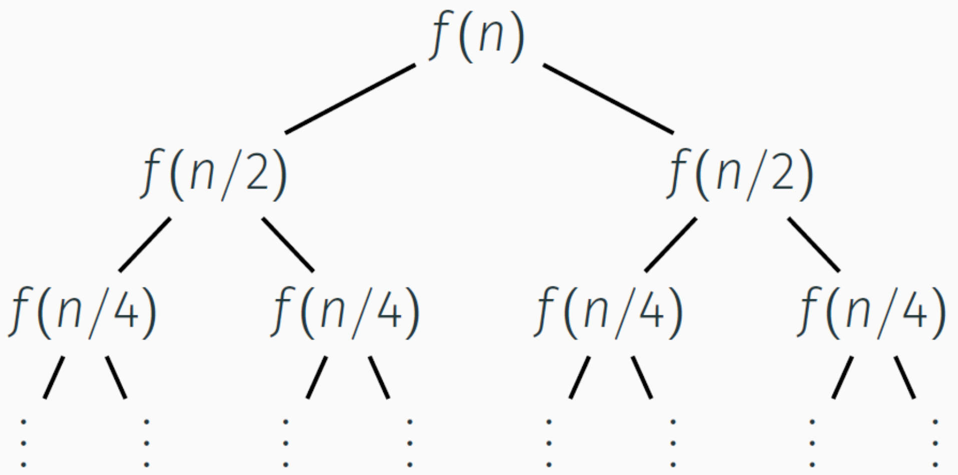
$$\begin{tikzcd} & f(n) \arrow[dl, dash] \arrow[dr, dash] & \\ T(n/2) & & T(n/2) \end{tikzcd}$$

Second unfolding: for each size-$n/2$ subproblem, cost of $f(n/2)$ plus cost of two subproblems of size $n/4$ each.

$$\begin{tikzcd} & & & f(n) \arrow[dll, dash] \arrow[drr, dash] & & & \\ & f(n/2) \arrow[dl, dash] \arrow[dr, dash] & & & & f(n/2) \arrow[dl, dash] \arrow[dr, dash] & \\ T(n/4) & & T(n/4) & & T(n/4) & & T(n/4) \end{tikzcd}$$

Unfolding the recurrence of Merge-Sort (cont'd)

Continue unfolding, until the problem size (= node label) gets down to 1:



In total, there are $\lg n + 1$ levels.

- Level 0 (root) has cost $C_0(n) = f(n)$.
- Level 1 has cost $C_1(n) = 2f(n/2)$.
- Level 2 has cost $C_2(n) = 4f(n/4)$.
- For $l < \lg n$, level l has cost $C_l(n) = 2^l f(n/2^l)$.
  Note that, since $d'\,n \le f(n) \le d\,n$, we have $d'\,n \le C_l(n) \le d\,n$.
- The last level (consisting of n leaves) has cost cn.

Analysing Merge-Sort with the recursion tree

The total cost of the algorithm is the sum of the costs of all levels:

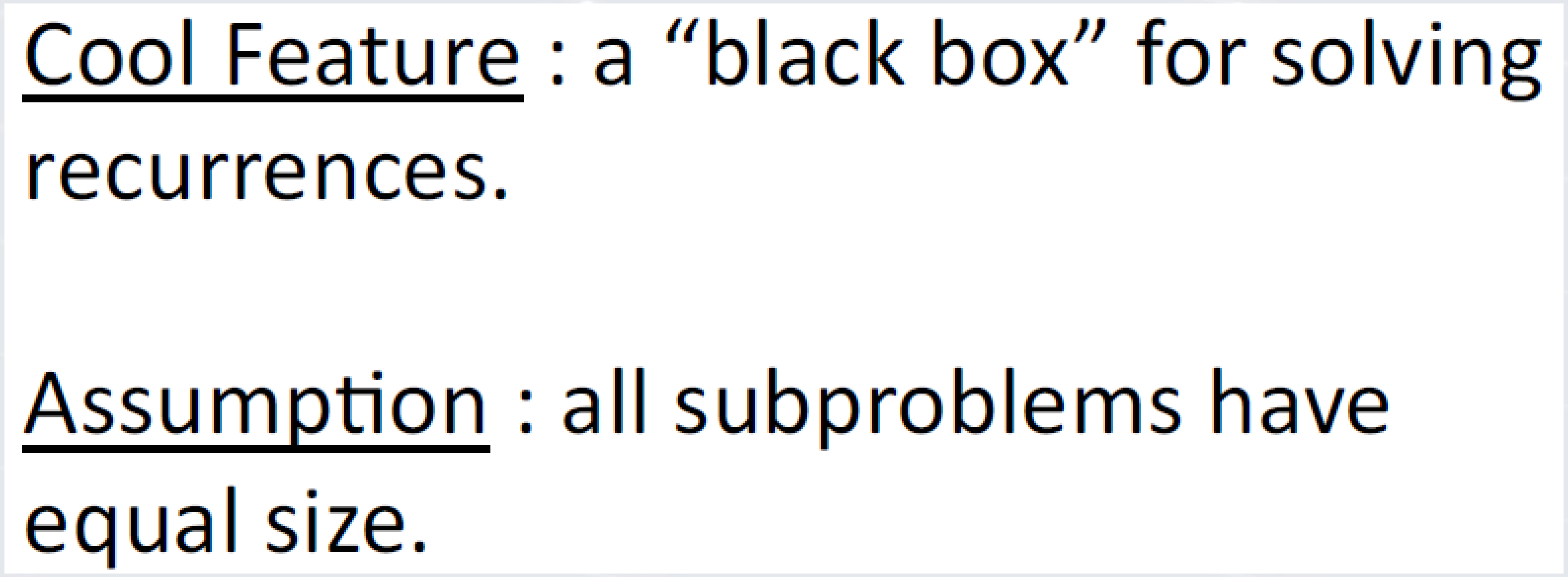
$$T(n) = \sum_{l=0}^{\lg n-1} C_l(n) + c\,n\,.$$

Using the relation $d'\,n \le C_l(n) \le dn$ for $l < \lg n$, we obtain the bounds

$$d'\,n\lg n + c\,n \le T(n) \le d\,n\lg n + c\,n\,.$$

Hence, $T(n) = \Theta(n \log n)$.

The Master Method



Cool Feature : a “black box” for solving recurrences.

Assumption : all subproblems have equal size.

The Master Method – recurrence format

1. Base Case : T(n) <= a constant for all sufficiently small n
2. For all larger n :

$$T(n) \leq aT(n/b) + O(n^d)$$

where

a = number of recursive calls (>= 1)

b = input size shrinkage factor (> 1)

d = exponent in running time of “combine step” (>=0)

[a,b,d independent of n]

The Master Method

Base doesn't matter (only changes leading constants)

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

Base matters

a = number of recursive calls (>= 1)

b = input size shrinkage factor (> 1)

d = exponent in running time of "combine step" (>=0)

The Master Method

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

Where are the respective values of a, b, d for a binary search of a sorted array, and which case of the Master Method does this correspond to?

- ○ 1, 2, 0 [Case 1]
- ○ 1, 2, 1 [Case 2]
- ○ 2, 2, 0 [Case 3]
- ○ 2, 2, 1 [Case 1]

<u>Merge Sort</u>

a = 2
b = 2
d = 1

$$b^d = \boxed{\text{a}} \Rightarrow \quad Case\ 1$$

$$T(n) = O(n^d \log n) = O(n \log n)$$

<u>Integer Multiplication Algorithm # 1</u>

a = 4
b = 2
d = 1

$$b^d = 2 < a \quad (Case \;\; 3)$$

$$\Rightarrow T(n) = O(n^{\log_b a}) = O(n^{\log_2 4})$$

$$= O(n^2)$$

<u>Strassen's Matrix Multiplication Algorithm</u>

a = 7
b = 2
d = 2

$$b^d = 4 < a \quad (Case \;\; 3)$$

$$\Rightarrow T(n) = O(n^{\log_2 7}) = O(n^{2.81})$$

=> beats the naïve iterative algorithm !

The Master Method

If $T(n) \le aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

Where are the respective values of a, b, d for a binary search of a sorted array, and which case of the Master Method does this correspond to?

- ○ 1, 2, 0 [Case 1] $a = b^d \Rightarrow T(n) = O(n^d \log n) = O(\ \log n)$
- ○ 1, 2, 1 [Case 2]
- ○ 2, 2, 0 [Case 3]
- ○ 2, 2, 1 [Case 1]

Merge Sort

a = 2
b = 2
d = 1

$b^d = \boxed{a} \Rightarrow \ Case\ 1$

$T(n) = O(n^d \log n) = O(n \log n)$

Integer Multiplication Algorithm # 1

a = 4
b = 2
d = 1

$b^d = 2 < a \quad (Case\ \ 3)$

$\Rightarrow T(n) = O(n^{\log_b a}) = O(n^{\log_2 4})$

$= O(n^2)$

Strassen's Matrix Multiplication Algorithm

a = 7
b = 2
d = 2

$b^d = 4 < a \quad (Case\ \ 3)$

$\Rightarrow T(n) = O(n^{\log_2 7}) = O(n^{2.81})$

=> beats the naïve iterative algorithm !

The Master Method – proof Case 1

If $T(n) \le aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

Assume : recurrence is

I. $T(1) \le c$

II. $T(n) \le aT(n/b) + cn^d$

(For some constant c)

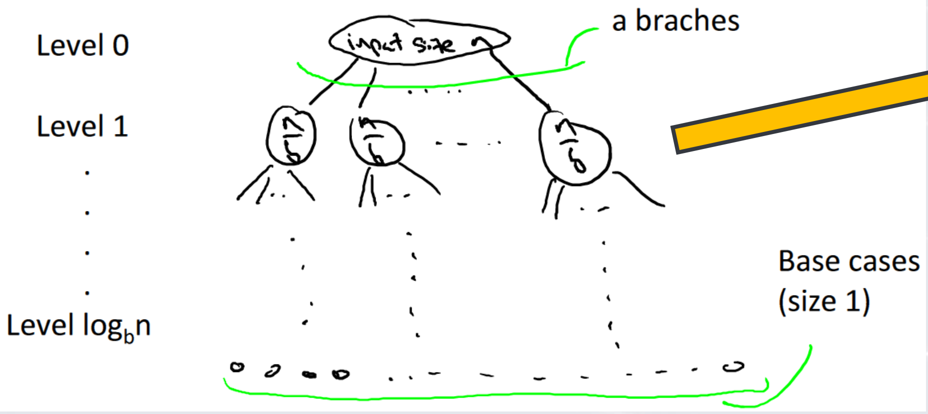
And n is a power of b.

(general case is similar, but more tedious)

Idea : generalize MergeSort analysis.

(i.e., use a recursion tree)

The Recursion Tree



Work at a Single Level

Total work at level j [ignoring work in recursive calls]

$$\le \underbrace{a^j}_{\text{\# of level-j subproblems}} \cdot c \cdot \overbrace{\Big(\underbrace{\frac{n}{b^j}}_{\text{Size of each level-j subproblem}}\Big)^d}^{\text{Work per level-j subproblem}} = cn^d \cdot \left(\frac{a}{b^d}\right)^j$$

Total Work

Summing over all levels j = 0,1,2,..., $\log_b n$:

Total work $\le cn^d \cdot \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j$

a = rate of subproblem proliferation (RSP)

b^d = rate of work shrinkage (RWS) (per subproblem)

The Master Method – proof Case 1

If $T(n) \le aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

Total Work

Summing over all levels j = 0,1,2,..., $\log_b$n :

$$\text{Total work} \le cn^d \cdot \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j$$

a = rate of subproblem proliferation (RSP)

b^d = rate of work shrinkage (RWS)

(per subproblem)

$$\text{Total work:} \quad \le cn^d \times \underbrace{\sum_{j=0}^{\log_b n} \boxed{\left(\frac{a}{b^d}\right)^j}}_{=\ (\log_b n + 1)} \qquad (*)$$

= 1 for all j

=1

$$If \quad a = b^d, \ then$$

$$(*) = cn^d(\log_b n + 1)$$

$$= O(n^d \log n)$$

[end Case 1]

When NOT to use Divide-and-Conquer

- Determining Thresholds
  - What is the optimal threshold value of n ? (n < 591, n < 128)

- If possible we should avoid divide-and-conquer in following
  - An instance of size n is divided into two or more instances each almost of size n
    - LEADS TO EXPONENTIAL-TIME ALGORITHM
  - An instance of size n is divided into almost n instances of size n/c, where c is a constant
    - LEADS TO $n^{\Theta(lg\ n)}$ ALGORITHM