COMP 6651
Algorithm Design Techniques

Lecturer: Thomas Fevens

Department of Computer Science and Software Engineering, Concordia U
thomas.fevens@concordia.ca

Table of Contents

1 D&C algorithms
2 Binary Search
3 Merge-Sort
4 Integer Mult.
5 Multiplying Sq. Matrices
6 Max. Subarray Prob.
7 Quicksort
8 Lower bound

One very basic type of algorithms that was seen in an elementary algorithms course:

Divide-and-conquer algorithms

Divide the problem into subproblems

Conquer each subproblem by solving them recursively (small size subproblems are solved directly)

Combine the solutions to the subproblems into a solution of the original problem

Design issues:
- How many subproblems we divide into,
- what are the "small" sizes solved directly, and
- how to combine solutions of subproblems into a solution of the original problem depends on each individual problem.

Analyzing run-time of divide-and-conquer algorithms

Assume we have a problem of size n.

In most cases, when a subproblem is of size $\leq c$, it takes a constant time:

$$T(n) = \Theta(1) \text{ if } n \leq c$$

Assume $n > c$, and the problem can be divided into a instances of the same problem of size $1/b$ of the original size (i.e., a subproblems of size n/b).

There can be some cost involved in breaking a problem into subproblems: $D(n)$

There can be some cost involved in combining solutions of subproblems into a solution of the problem: $C(n)$

$$T(n) = aT(n/b) + D(n) + C(n) \text{ if } n > c$$

Examples: a) Binary search in a sorted array

BinSearch(x, A, i, j)
\\ Search where x is in array A[i] to A[j]
  if i $>$ j **then return** -1
  mid = (i+j)/2
  if x $<$ A[mid] **then return BinSearch**(x,A,i,mid-1)
  else
    if x $>$ A[mid] **then return BinSearch**(x,A,mid+1,j)
    else
      return mid

Analysis of run-time for **BinSearch**:

$a = 1$, $b = 2$,

$D(n) = \Theta(1)$,

$C(n) = \Theta(1)$.

Then

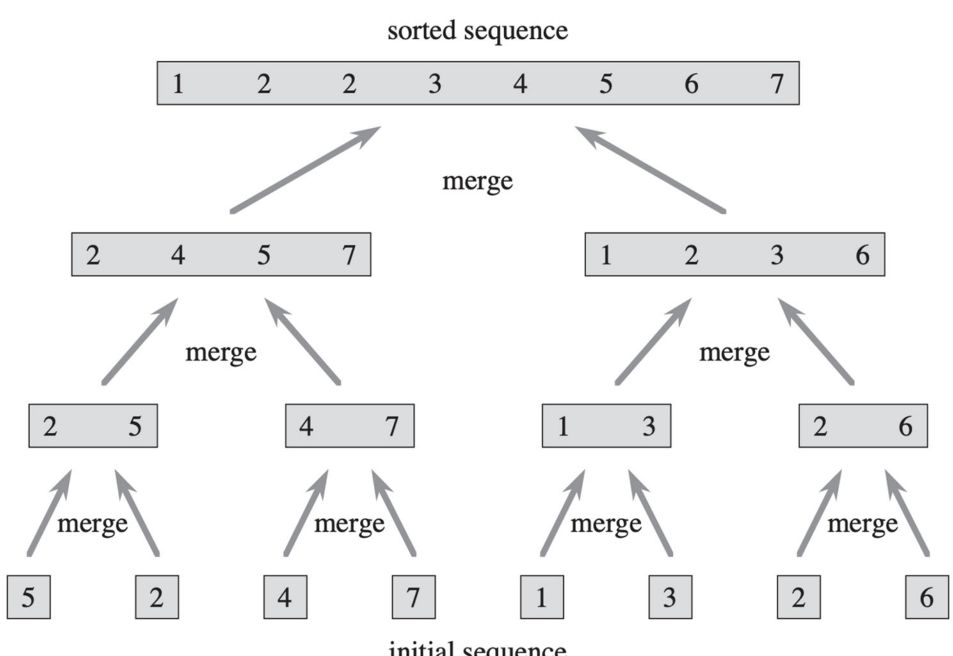
$$\begin{aligned} T(n) &= aT(n/b) + D(n) + C(n) \text{ if } n > c \\ &= T(n/2) + \Theta(1) + \Theta(1) \text{ if } n \geq 1 \\ &= T(n/2) + \Theta(1) \text{ if } n \geq 1 \\ &= T(n/2) + c \text{ if } n \geq 1 \\ &= \Theta(\log n) \end{aligned}$$

b) Merge-Sort (§2.3)

Merge-Sort(A, p, r)
\\\\ Sort A[p] to A[r]
  if p $<$ r **then**
    mid = (p+r)/2
    Merge-Sort(A,p,mid)
    Merge-Sort(A,mid,r)
    Merge(A,p,mid,r)

Worst case analyses:
$a = 2$,
$b = 2$,
$D(n) = \Theta(1)$,
$C(n) = \Theta(n)$,
$T(1) = \Theta(1)$.

$T(n) = 2T(n/2) + \Theta(n)$ if $n > 1$
$T(n) = 2T(n/2) + cn$ if $n > 1$
$T(n) = \Theta(n \log n)$



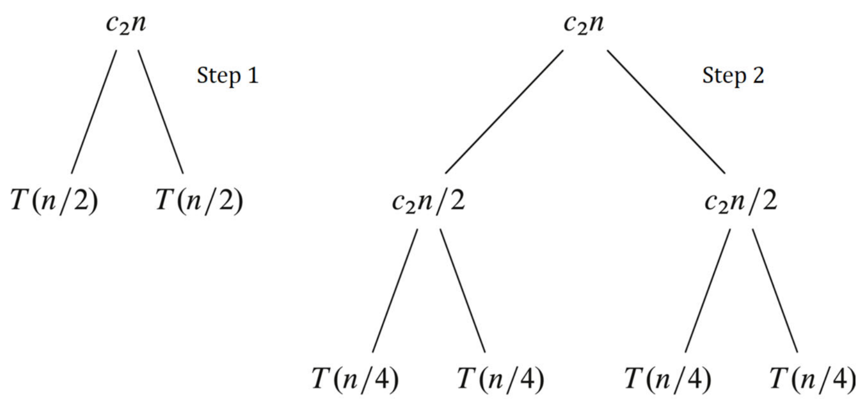
Note: Using recursion tree to understand recurrence

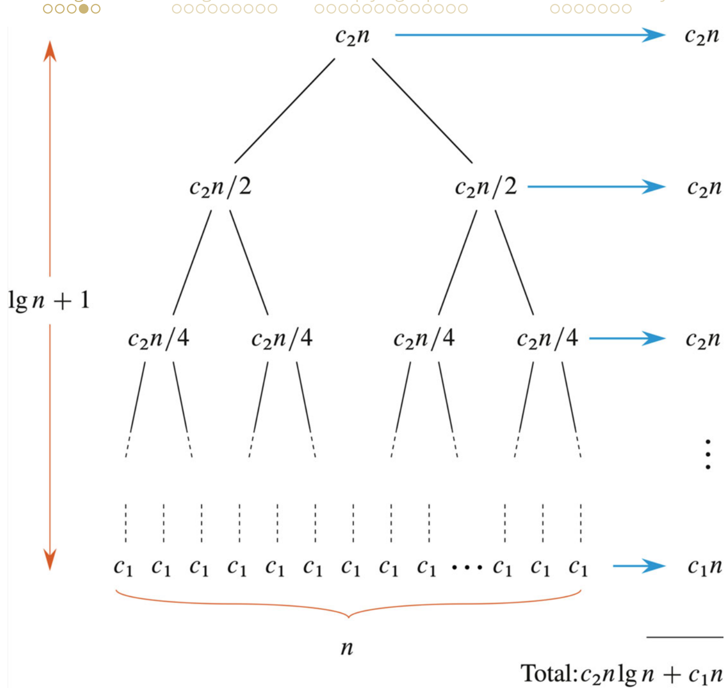
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

We rewrite the recurrence as

$$T(n) = \begin{cases} c_1 & \text{if } n = 1, \\ 2T(n/2) + c_2 n & \text{if } n > 1 \end{cases}$$

Recurrence steps:





Disadvantages of merge sort:

- Needs additional space when **Merge**(A, p, mid, r) is executed.
- It is not in-place sorting.

Advantages of merge sort:

- Worst case is quite the same as the best case.
- Run-time $\Theta(n \log n)$ is guaranteed.

c) Integer Multiplication (Not in CLRS 4th Ed.)

Input: X, Y – two n-digit integers
Output: $X \cdot Y$

Example:

$$X = 4512354$$
$$Y = 1238970$$
$$X \cdot Y = 5590671235380$$
$$(n = 7)$$

X	Y	$X \cdot Y$
9	9	81
99	99	9801
999	999	998001
9999	9999	99980001
99999	99999	9999800001

Observation: if X and Y are n-digit numbers then $X \cdot Y$ is at most a $2n$-digit number

Multiply($X[1..n]$, $Y[1..n]$)
  $Z[1..2n] \leftarrow 0$
  for $i \leftarrow n, 1$ **do**
    $carry \leftarrow 0$
    for $j \leftarrow n, 1$ **do**
      $m \leftarrow Z[i+j] + carry + X[j] \cdot Y[i]$
      $Z[i+j] \leftarrow m \mod 10$
      $carry \leftarrow \left\lfloor \frac{m}{10} \right\rfloor$
    $Z[i] \leftarrow carry$
  return Z

```
        2 6 4 2
      × 5 8 2 1
      ---------
        2 6 4 2
      5 2 8 4
  2 1 1 3 6
1 3 2 1 0
---------------
1 5 3 7 9 0 8 2
```

$X = [2, 6, 4, 2]$
$Y = [5, 8, 2, 1]$
$Z = [1, 5, 3, 7, 9, 0, 8, 2]$

Multiply$(X[1..n], Y[1..n])$
  $Z[1..2n] \leftarrow 0$
  for $i \leftarrow n, 1$ **do**
    $carry \leftarrow 0$
    for $j \leftarrow n, 1$ **do**
      $m \leftarrow Z[i+j] + carry + X[j] \cdot Y[i]$
      $Z[i+j] \leftarrow m \mod 10$
      $carry \leftarrow \left\lfloor \frac{m}{10} \right\rfloor$
    $Z[i] \leftarrow carry$
  return Z

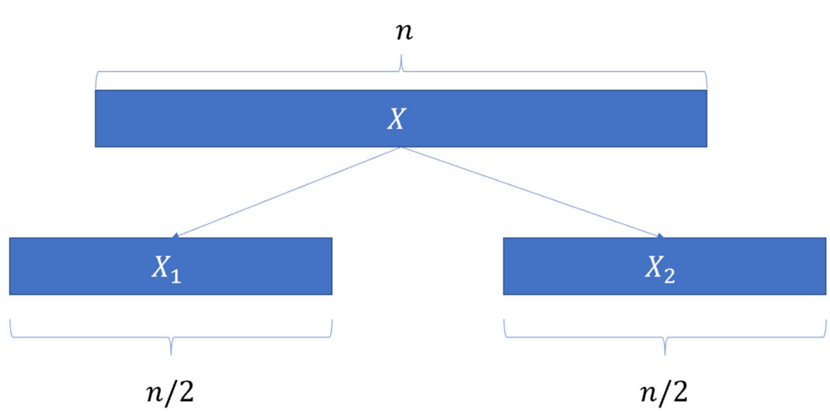
Cost measure: number of single-digit multiplications

$M(n)$ = worst-case cost of **Multiply** on inputs of length n

$M(n) = \Theta(n^2)$

Can we multiply two integers faster?
In 1960s, Kolmogorov conjectured NO
Karatsuba disproved the conjecture
Karatsuba's idea: divide and conquer!

$$X = 10^{n/2} X_1 + X_2$$
$$Y = 10^{n/2} Y_1 + Y_2$$



$$X \cdot Y = \left(10^{n/2}X_1 + X_2\right) \cdot \left(10^{n/2}Y_1 + Y_2\right)$$
$$= 10^n X_1 \cdot Y_1 + 10^{\frac{n}{2}}(X_1 \cdot Y_2 + X_2 \cdot Y_1) + X_2 \cdot Y_2$$

Multiply($X[1..n]$, $Y[1..n]$)
  if $n = 1$ **then**
    return $X \cdot Y$
  $R_1 \leftarrow$ **Multiply**(X_1, Y_1)
  $R_2 \leftarrow$ **Multiply**(X_1, Y_2)
  $R_3 \leftarrow$ **Multiply**(X_2, Y_1)
  $R_4 \leftarrow$ **Multiply**(X_2, Y_2)
  return $10^n R_1 + 10^{\frac{n}{2}}(R_2 + R_3) + R_4$

$M(n)$ = number of single-digit multiplications in this procedure

$M(n) = 4M\left(\frac{n}{2}\right) + \Theta(1)$
$M(1) = \Theta(1)$

Solves to $M(n) = \Theta(n^2)$
(See Master's Theorem)

So, no improvement...

Idea:

$$X \cdot Y = 10^n X_1 \cdot Y_1 + 10^{\frac{n}{2}}(X_1 \cdot Y_2 + X_2 \cdot Y_1) + X_2 \cdot Y_2$$

We don't need $X_1 \cdot Y_2$ and $X_2 \cdot Y_1$ to be computed separately
We only need $W = X_1 \cdot Y_2 + X_2 \cdot Y_1$
Can we computer W with one extra recursive call?

$$(X_1 - X_2) \cdot (Y_1 - Y_2) = X_1 \cdot Y_1 - (X_1 \cdot Y_2 + X_2 \cdot Y_1) + X_2 \cdot Y_2$$

  $R_1 \leftarrow$ **Multiply**(X_1, Y_1)
  $R_2 \leftarrow$ **Multiply**(X_2, Y_2)
  $R_3 \leftarrow$ **Multiply**($X_1 - X_2, Y_1 - Y_2$)

Then: $W = R_1 + R_2 - R_3$

Multiply$(X[1..n], Y[1..n])$
  if $n = 1$ **then**
    return $X \cdot Y$
  $R_1 \leftarrow$ **Multiply**(X_1, Y_1)
  $R_2 \leftarrow$ **Multiply**(X_2, Y_2)
  $R_3 \leftarrow$ **Multiply**$(X_1 - X_2, Y_1 - Y_2)$
  return $10^n R_1 + 10^{\frac{n}{2}}(R_1 + R_2 - R_3) + R_2$

$M(n)$ = number of single-digit multiplications in this procedure

$M(n) = 3M\left(\frac{n}{2}\right) + \Theta(1)$
$M(1) = \Theta(1)$

Solves to

$$M(n) = \Theta(n^{\log_2 3}) = O(n^{1.585})$$

(See Master's Theorem)

D&C algorithms | Binary Search | Merge-Sort | Integer Mult. | Multiplying Sq. Matrices | Max. Subarray Prob. | Quicksort | Lower bound

Notes

1. Actual runtime also includes additions, copying arrays, and shifting arrays.
   $T(n)$ = worst-case runtime
   $T(n) = 3T\left(\frac{n}{2}\right) + O(n)$
   $T(1) = \Theta(1)$
   Still solves to $T(n) = \Theta(n^{\log_2 3})$

2. What if n is not divisible by 2?
   There exists $n \leq n' \leq 2n$ such that n' is a power of 2
   $T(n) \leq T(n') = O\left((n')^{\log_2 3}\right) = O\left((2n)^{\log_2 3}\right) = O(n^{\log_2 3})$

d) Multiplying Square Matrices (§4.1-4.2)

Input: Three $n \times n$ (square) matrices, $A = (a_{ij})$, $B = (b_{ij})$, and $C = (c_{ij})$.
Result: The matrix product $A \cdot B$ is added into C, so that

$c_{ij} = c_{ij} + \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$
for $i, j = 1, 2, \ldots, n$.
If only the product $A \cdot B$ is needed, then zero out all entries of C beforehand.

Straightforward method

Matrix-Multiply(A, B, C, n)

```
for i ← 1, n do
    for j ← 1, n do
        for k ← 1, n do
            c_ij ← c_ij + a_ik · b_kj
```


Simple Divide-and-Conquer Algorithm

For simplicity, assume that C is initialized to 0, so computing $C = A \cdot B$.

If $n > 1$, partition each of A, B, C into four $n/2 \times n/2$ matrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Rewrite $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

giving the four equations

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{aligned}$$

Each of these equations multiplies two $n/2 \times n/2$ matrices and then adds their $n/2 \times n/2$ products. Assume that n is an exact power of 2, so that submatrix dimensions are always integer.

Use these equations to get a divide-and-conquer algorithm:

Matrix-Multiply-Recursive(A, B, C, n)

if $n = 1$ **then** ▷ Base case.
    $c_{11} \leftarrow c_{11} + a_{11} \cdot b_{11}$
    return
partition A, B, and C into ▷ Divide.
$n/2 \times n/2$ submatrices A_{ij}, B_{ij}, C_{ij}, $i, j = 1, 2$
▷ Conquer.
Matrix-Multiply-Recursive$(A_{11}, B_{11}, C_{11}, n/2)$
Matrix-Multiply-Recursive$(A_{11}, B_{12}, C_{12}, n/2)$
Matrix-Multiply-Recursive$(A_{21}, B_{11}, C_{21}, n/2)$
Matrix-Multiply-Recursive$(A_{21}, B_{12}, C_{22}, n/2)$
Matrix-Multiply-Recursive$(A_{12}, B_{21}, C_{11}, n/2)$
Matrix-Multiply-Recursive$(A_{12}, B_{22}, C_{12}, n/2)$
Matrix-Multiply-Recursive$(A_{22}, B_{21}, C_{21}, n/2)$
Matrix-Multiply-Recursive$(A_{22}, B_{22}, C_{22}, n/2)$

Aside:

The book briefly discusses the question of how to avoid copying entries when partitioning matrices. Can partition matrices without copying entries by instead using index calculations.

Analysis

Let $T(n)$ be the time to multiply two $n \times n$ matrices.

Base case: $n = 1$. Perform one scalar multiplication: $\Theta(1)$.

Recursive case: $n > 1$.

- Dividing takes $\Theta(1)$ time, using index calculations. [Otherwise, $\Theta(n^2)$ time.]
- Conquering makes 8 recursive calls, each multiplying $n/2 \times n/2$ matrices $\Longrightarrow 8T(n/2)$.
- No combine step, because C is updated in place.

Recurrence (omitting the base case) is $T(n) = 8T(n/2) + \Theta(1)$. Can use master method to show that it has solution $T(n) = \Theta(n^3)$.

Bushiness of recursion trees:

Compare this recurrence with the **Merge-Sort** recurrence $T(n) = 2T(n/2) + \Theta(n)$. If we draw out the recursion trees, the factor of 2 in the merge-sort recurrence says that each non-leaf node has 2 children.

But the factor of 8 in the recurrence $T(n) = 8T(n/2) + \Theta(1)$ for **Matrix-Multiply-Recursive** says that each non-leaf node has 8 children. Get a bushier tree with many more leaves, even though internal nodes have a smaller cost ($\Theta(1)$ versus $\Theta(n)$).

Strassen's Algorithm

Idea: Make the recursion tree less bushy. Perform only 7 recursive multiplications of $n/2 \times n/2$ matrices, rather than 8. Will cost several additions/subtractions of $n/2 \times n/2$ matrices.

Since a subtraction is a "negative addition," just refer to all additions and subtractions as additions.

Example of reducing multiplications: Given x and y, compute $x^2 - y^2$. Obvious way uses 2 multiplications and one subtraction. But observe:

$$x^2 - y^2 = x^2 - xy + xy - y^2 = x(x - y) + y(x - y) = (x + y)(x - y)$$

So, at the expense of one extra addition, can get by with only 1 multiplication. Not a big deal if x, y are scalars, but can make a difference if they are matrices.

The algorithm:

1. If $n = 1$, the matrices each contain a single element. Perform a single scalar multiplication and a single scalar addition, as in line 2 of **Matrix-Multiply-Recursive**, taking $\Theta(1)$ time, and return.
2. When $n > 1$, partition the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in line 2 of **Matrix-Multiply-Recursive**. This step takes $\Theta(1)$ time by index calculation, just as in **Matrix-Multiply-Recursive**.
3. Create $n/2 \times n/2$ matrices $S_1, S_2, \ldots, S_{10}$, each of which is the sum or difference of two submatrices from steps 1 and 2. Create and zero the entries of seven $n/2 \times n/2$ matrices $P_1, P_2, \ldots, P_7$ to hold seven $n/2 \times n/2$ matrix products. All 17 matrices can be created, and the P_i initialized, in $\Theta(n^2)$ time.

The algorithm, cont.:

4. Using the submatrices from steps 1 and 2 and the matrices $S_1, S_2, \ldots, S_{10}$ created in step 3, recursively compute each of the seven matrix products $P_1, P_2, \ldots, P_7$, taking $7T(n/2)$ time.
5. Update the four submatrices C_{11}, C_{12}, C_{21}, C_{22} of the result matrix C by adding or subtracting various P_i matrices, which takes $\Theta(n^2)$ time.

Analysis

Recurrence will be $T(n) = 7T(n/2) + \Theta(n^2)$. By the master method, solution is $T(n) = \Theta(n^{\lg 7})$. Since $\lg 7 < 2.81$, the running time is $O(n^{2.81})$, beating the $\Theta(n^3)$-time algorithms.

Details

Step 3: Create the 10 matrices

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} , \\
S_2 &= A_{11} + A_{12} , \\
S_3 &= A_{21} + A_{22} , \\
S_4 &= B_{21} - B_{11} , \\
S_5 &= A_{11} + A_{22} , \\
S_6 &= B_{11} + B_{22} , \\
S_7 &= A_{12} - A_{22} , \\
S_8 &= B_{21} + B_{22} , \\
S_9 &= A_{11} - A_{21} , \\
S_{10} &= B_{11} + B_{12} .
\end{aligned}
$$

Add or subtract $n/2 \times n/2$ matrices 10 times $\Rightarrow$ time is $\Theta(n^2)$.

D&C algorithms Binary Search Merge-Sort Integer Mult. Multiplying Sq. Matrices Max. Subarray Prob. Quicksort Lower bound

Details, cont.

Step 4: Compute the 7 matrices

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22} , \\
P_2 &= S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22} , \\
P_3 &= S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11} , \\
P_4 &= A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11} , \\
P_5 &= S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}, \\
P_6 &= S_7 \cdot S_8 = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}, \\
P_7 &= S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} .
\end{aligned}
$$

The only multiplications needed are in the middle column; right-hand column just shows the products in terms of the original submatrices of A and B.

Details, cont.

Step 5: Add and subtract the P_i to construct submatrices of C:

$$
\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 \,, \\
C_{12} &= P_1 + P_2 \,, \\
C_{21} &= P_3 + P_4 \,, \\
C_{22} &= P_5 + P_1 - P_3 - P_7 \,.
\end{aligned}
$$

Example

Example of how C_{11} is reconstructed using additions and the previously defined P and S matrices. Recall definition of C_{11}: $C_{11} = P_5 + P_4 - P_2 + P_6$. Expanding the right-hand side:

$$
\begin{array}{lllllll}
A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} & & & & \\
\qquad\qquad - A_{22} \cdot B_{11} & + A_{22} \cdot B_{21} & & & \\
\quad - A_{11} \cdot B_{22} & & - A_{12} \cdot B_{22} & & \\
\qquad\qquad\qquad - A_{22} \cdot B_{22} & - A_{22} \cdot B_{21} & + A_{12} \cdot B_{22} & + A_{12} \cdot B_{21} \\
\hline
A_{11} \cdot B_{11} & & & + A_{12} \cdot B_{21}
\end{array}
$$

All four examples are fully worked out in the text.

Notes

Strassen's algorithm was the first to beat $\Theta(n^3)$ time, but it's not the asymptotically fastest known. A method by Coppersmith and Winograd runs in $\Theta(n^{2.376})$ time. Current best asymptotic bound (not practical) is $\Theta(n^{2.37286})$.

Practical issues against Strassen's algorithm:

- Higher constant factor than the obvious $\Theta(n^3)$-time method.
- Not good for sparse matrices.
- Not numerically stable: larger errors accumulate than in the obvious method.
- Submatrices consume space, especially if copying.

e) Maximum Subarray Problem (§4.1 of 3rd Ed. of CLRS; not in 4th Ed.)

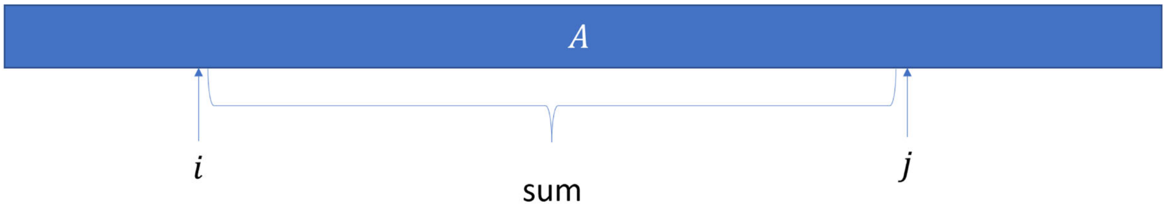
Input: $A[1..n]$ - array of n integers
Output: S - maximum sum of a contigous subarray, i.e., there exists $1 \leq i < j \leq n$ such that $S = \sum_{k=i}^{j} A[k]$ and S is maximized

Example:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

maximum subarray

Naive Algorithm

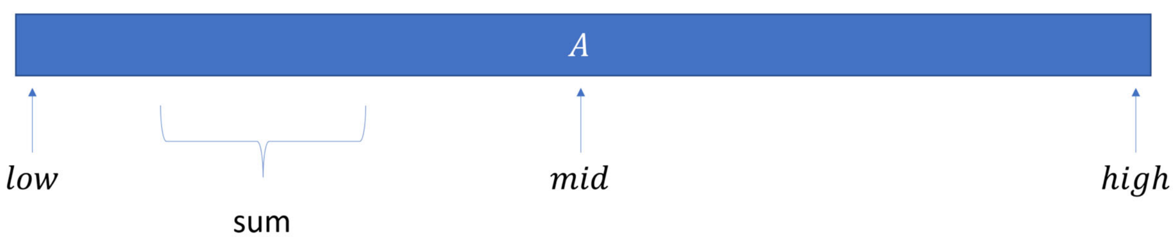


Check every pair of indices $1 \leq i < j \leq n$

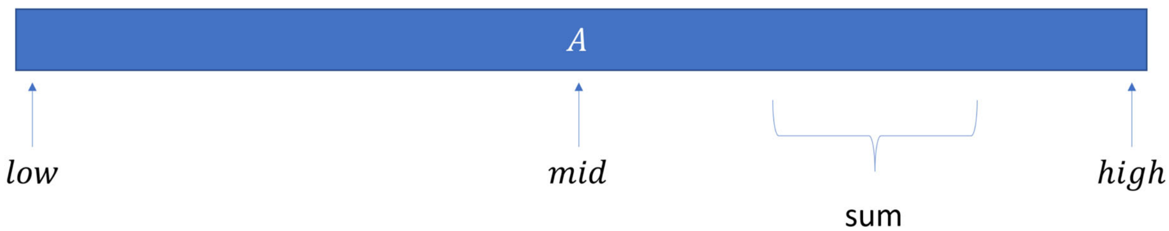
Even if we can compute each such sum in constant time there are still $\binom{n}{2} = \Theta(n^2)$ such pairs of indices i and j

Naive algorithm runs in time $\Omega(n^2)$

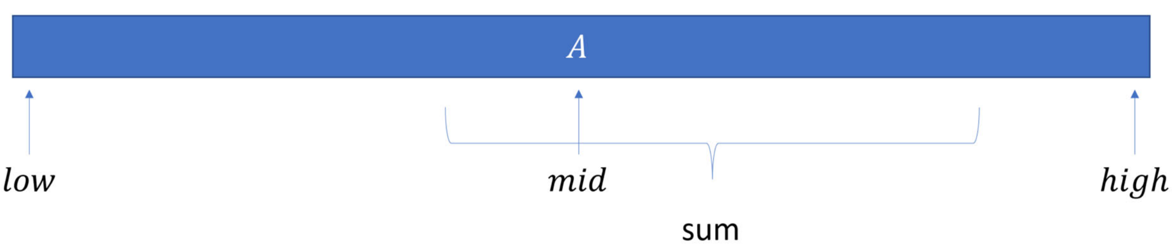
A Divide and Conquer Algorithm



If maximum subarray $A[i..j]$ doesn't cross mid
then it entirely lies in $A[low..mid]$



If maximum subarray $A[i..j]$ doesn't cross *mid*
then it entirely lies in $A[low..mid]$
or it entirely lies in $A[mid + 1..high]$



If maximum subarray $A[i..j]$ doesn't cross *mid*
then it entirely lies in $A[low..mid]$
or it entirely lies in $A[mid + 1..high]$
OR maximum subarray crosses *mid*

MaxCrossingSubarray($A, low, mid, high$)

$L \leftarrow -\infty$; $R \leftarrow -\infty$
$S \leftarrow 0$
for $i \leftarrow mid, low$ **do** ▷ find max sum to left starting at $A[mid]$
  $S \leftarrow S + A[i]$
  $L \leftarrow \max(L, S)$
$S \leftarrow 0$
for $i \leftarrow mid + 1, high$ **do** ▷ find max sum to right starting at $A[mid + 1]$
  $S \leftarrow S + A[i]$
  $R \leftarrow \max(R, S)$
return $L + R$

Observe that the body of the function does $O(n)$ work.

MaxSubarray($A, mid, high$)

if $high = low + 1$ **then**
  return $A[low] + A[high]$
if $high \leq low$ **then**
  return $-\infty$
$mid \leftarrow \left\lfloor \frac{low+high}{2} \right\rfloor$
$left \leftarrow$ **MaxSubarray**(A, low, mid)
$right \leftarrow$ **MaxSubarray**($A, mid + 1, high$)
$cross \leftarrow$ **MaxCrossingSubarray**($A, low, mid, high$)
return $\max(left, cross, right)$

Initial call: **MaxSubarray**($A, 1, n$)

$T(n)$ = worst-case runtime on instances of length n

MaxSubarray on input of length n:
- make 2 recursive calls on inputs of size $n/2$
- does additional $O(n)$ work

Thus, $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$
Base cases: $T(0)$, $T(1)$, $T(2) = O(1)$

Therefore, $T(n) = O(n \log n)$

e) Quick-Sort: (§7)

Quicksort is based on the three-step process of divide-and-conquer.

To sort the subarray $A[p..r]$:

Divide: Partition $A[p..r]$ in two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$, such that each element in the first subarray $A[p..q-1]$ is $\leq A[q]$ and $A[q]$ is $<$ each element in the second subarray $A[q+1..r]$.

Conquer: Sort the two subarrays by recursive calls to **Quicksort**.

Combine: No work is needed to combine the subarrays, because they are sorted *in place*.

The Divide step is performed using a procecure **Partition**, which returns the index q that marks the position separating the subarrays.

Quicksort(A, p, r)
\\\\ Sort A[p] to A[r]
  if p $<$ r **then** ▷ at least 2 values
    q =**Partition**(A,p,r)
    \\\\ partition A into two subarrays
    \\\\ such that A[i] $\leq$ A[q] for i $<$ q
    \\\\ and A[q] $<$ A[j] for q $<$ j
    Quicksort(A,p,q-1)
    Quicksort(A,q+1,r)

Best case:
$a = 2$,
$b = 2$,
$D(n) = \Theta(n)$,
$C(n) = \Theta(1)$.

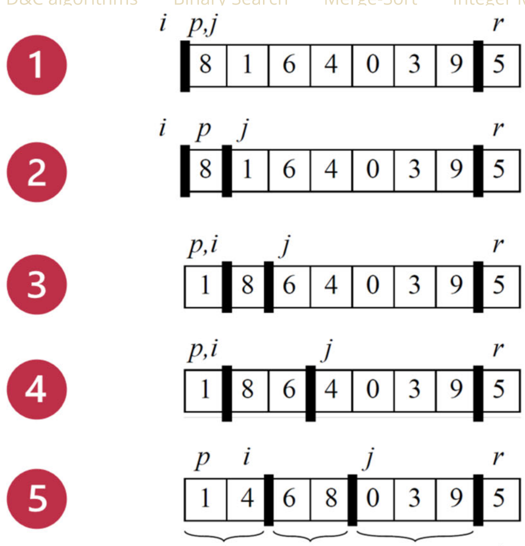
$T(n) = 2T(n/2) + \Theta(n)$ if $n \geq 1$
$T(n) = 2T(n/2) + cn$ if $n \geq 1$
$T(n) = \Theta(n \log n)$

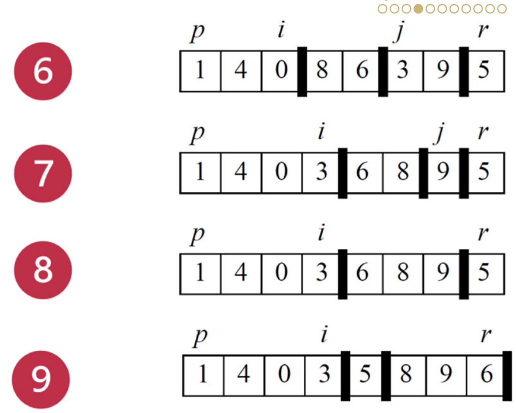
Partitioning

Partitioning of the subarray $A[p..r]$ is done by selection one element of the array A as a **pivot** and this element splits that array into two parts.

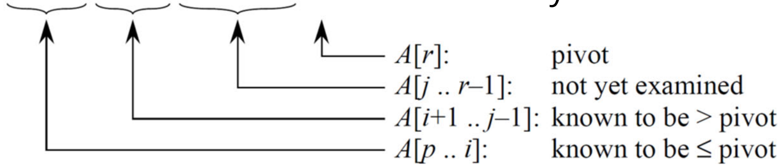
Partition(A, p, r)

$x \leftarrow A[r]$ \\ the last element is selected as the pivot
$i \leftarrow p - 1$
for $j \leftarrow p, r - 1$ **do**
  if $A[j] \leq x$ **then** $\triangleright$ All elements $\leq$ pivot moved to the front
    $i \leftarrow i + 1$
    exchange $A[i]$ with $A[j]$
exchange $A[i + 1]$ with $A[r]$
return $i + 1$ \\ new index of pivot





Time to partition: $\Theta(n)$ for n-element array.



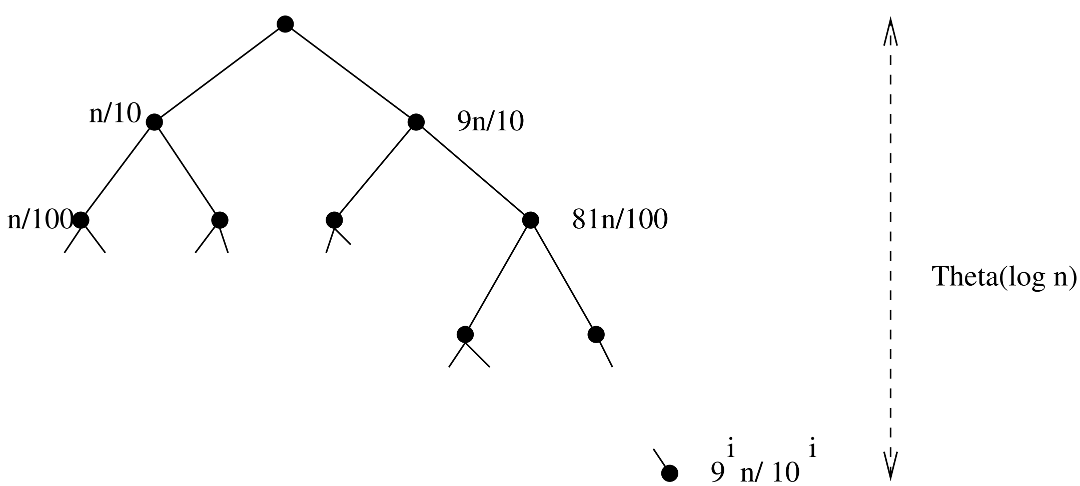
Worst case:

$a = 2$, (one of two subproblems is empty and the other is of size $n - 1$)
$D(n) = \Theta(n)$,
$C(n) = \Theta(1)$.

$T(n) = T(n-1) + T(0) + cn$ if $n \geq 1$
$T(n) = \Theta(n^2)$

What is the **average case**?

Even when the split produces $9/10$ elements in one subarray and $1/10$ in the other subarray *all the time*, the number of levels in the calls is bounded by $\Theta(\log n)$.



We need to repeat the splitting at most $\log_{\frac{10}{9}} n$ times.

Thus, $T(n) = \Theta(n \log n)$

How to guarantee that we get a “good split” often enough?

Randomized Quicksort

a) Select the pivot randomly:

In the partition function, generate a random number i, $p \leq i \leq r$ and swap $A[i]$ with $A[r]$ to be used as the pivot.

b) OR before applying Quicksort, permutate the elements of the array in random manner:

Assume **Random**(i, n) is a function that selects an integer between i and n with the same probability (e.g., a uniform distribution random function).

Randomize-In-Place(A)

$n \leftarrow$ *A.length()*
for $i \leftarrow 1, n$ **do**
  swap$(A[i], A[$**Random**$(i, n)])$

D&C algorithms Binary Search Merge-Sort Integer Mult. Multiplying Sq. Matrices Max. Subarray Prob. Quicksort Lower bound

In **Randomize-in-Place**:

No additional space is required.
Run time is $\Theta(n)$.

Does it produce a random permutation?

Yes, if **Random** is correct:

- Notice that once an element is swapped in position $A[i]$, it is not swapped again, the probability of any element being the first is $1/n$,
any of the remaining element has the same probability of being second, etc.

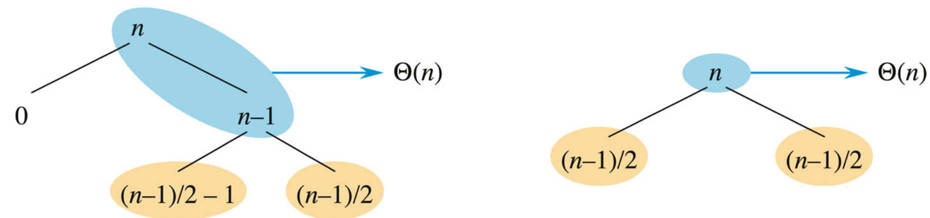
This procedure is often used for any algorithm when the average run-time of the algorithm is much better than the worst case.

Warning: It is easy to write an incorrect version of randomize!

Intuition for the Average Case

Splits in the recursion tree will not always be constant. There will usually be a mix of good and bad splits throughout the recursion tree.

To see that this doesn't affect the asymptotic running time of quicksort, assume that levels alternate between best-case and worst-case splits.



The extra level in the left-hand figure only adds to the constant hidden in the Θ-notation.

There are still the same number of subarrays to sort, and only twice as much work was done to get to that point.

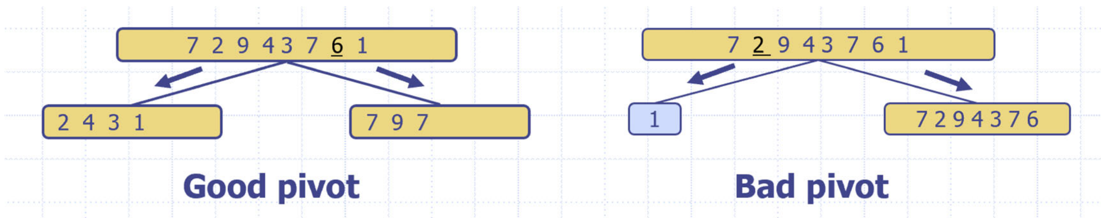
Theorem

The expected running time of randomized quicksort on a sequence of size n is $O(n \log n)$.

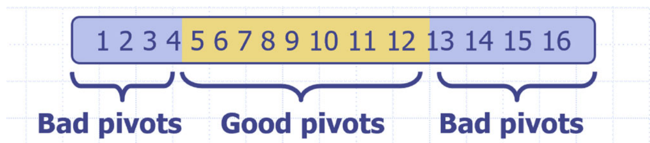
Proof
We use the following fact:
The expected number of times that a fair coin must be flipped until it shows "heads" k times is 2k.

In the randomized quicksort, the probability of getting a split of m elements with at least $m/4$ elements in one part and at most $3m/4$ elements in the other is $1/2$. (call this a good split using a good pivot)

Examples:



The probability of getting a good split is $1/2$.



The expected number of times splitting should be repeated to get a good split $\log_{\frac{4}{3}} n$ times is $2\log_{\frac{4}{3}} n$.

Expected run-time is $O(n \log n)$

Theorem (§8.1)

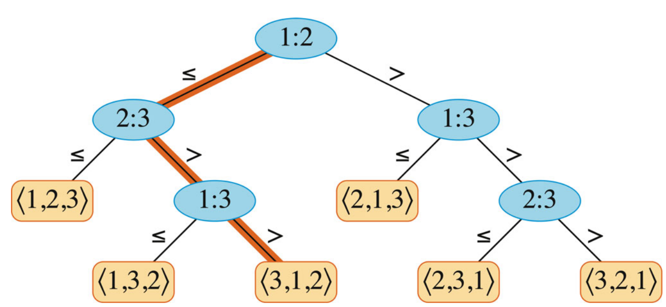
Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons to sort n elements in the worst case.

Proof

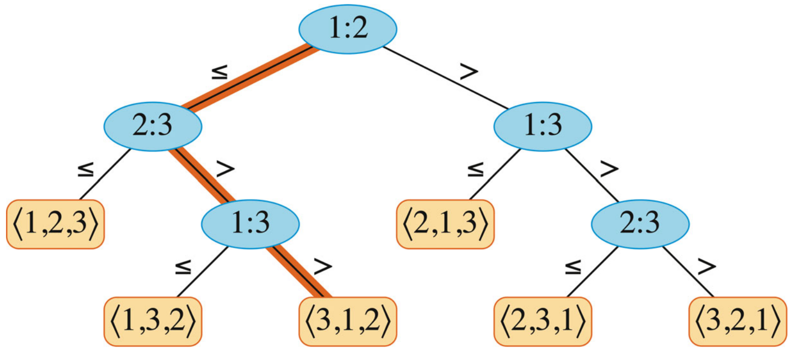
Consider a decision tree corresponding to a comparison-based sorting algorithm.

Each leaf of the decision tree corresponds to one of the permutations of the input.

A path from the root to a leaf correspond to a possible execution of the algorithm.



Proof, cont.



There are $n!$ possible permutations of n elements.

The decision tree must have $n!$ leafs,

its depth is $\geq \log n! \approx \log \sqrt{2\pi n}\frac{n}{e}^{n} e^{\frac{1}{12n}} = cn \log n$

Sorting faster than $\Theta(n \log n)$?

Are there some sorting algorithms that can “beat” the $\Theta(n \log n)$ barrier of comparison-based algorithms?

We can do better if we know something about either

- the distribution of elements to be sorted, or
- the elements to be sorted.

Radix-sort and similar algorithms (see the textbook if you are not familiar with them):

Their run-time to sort n elements is $\Theta(kn)$

where k is the maximal length of the *keys* used for sorting.

It is better when k is smaller than $\log n$ ($\log n$ is the minimum number of bits to represent n numbers).