Divide & Conquer

Jordi Cortadella and Jordi Petit
Department of Computer Science

Divide & Conquer

Jordi Cortadella and Jordi Petit
Department of Computer Science

Divide & Conquer

Jordi Cortadella and Jordi Petit
Department of Computer Science

Divide & Conquer

Jordi Cortadella and Jordi Petit
Department of Computer Science

Divide & Conquer

Jordi Cortadella and Jordi Petit
Department of Computer Science

Divide & Conquer

Jordi Cortadella and Jordi Petit
Department of Computer Science

Conventional product of polynomials

Example:

\[ P(x) = 2x^3 + x^2 - 4 \]
\[ Q(x) = x^2 - 2x + 3 \]

\[ (P \cdot Q)(x) = 2x^5 + (-4 + 1)x^4 + (6 - 2)x^3 + 8x - 12 \]
\[ (P \cdot Q)(x) = 2x^5 - 3x^4 + 4x^3 + 8x - 12 \]

Conventional product of polynomials

Example:

\[ P(x) = 2x^3 + x^2 - 4 \]
\[ Q(x) = x^2 - 2x + 3 \]

\[ (P \cdot Q)(x) = 2x^5 + (-4 + 1)x^4 + (6 - 2)x^3 + 8x - 12 \]
\[ (P \cdot Q)(x) = 2x^5 - 3x^4 + 4x^3 + 8x - 12 \]

Divide-and-conquer algorithms

• Strategy:
  – Divide the problem into smaller subproblems of the same type of problem
  – Solve the subproblems recursively
  – Combine the answers to solve the original problem

• The work is done in three places:
  – In partitioning the problem into subproblems
  – In solving the basic cases at the tail of the recursion
  – In merging the answers of the subproblems to obtain the solution of the original problem

Conventional product of polynomials

function PolynomialProduct(P, Q)
// P and Q are vectors of coefficients.
// Returns R = P \times Q.
// degree(P) = size(P)-1, degree(Q) = size(Q)-1.
// degree(R) = degree(P)+degree(Q).

R = vector with size(P)+size(Q)-1 zeros;

for each \( P_i \)
  for each \( Q_j \)
    \[ R_{i+j} = R_{i+j} + P_i \cdot Q_j \]

return R

Complexity analysis:
• Multiplication of polynomials of degree \( n \): \( O(n^2) \)
• Addition of polynomials of degree \( n \): \( O(n) \)
Product of polynomials: Divide&Conquer

Assume that we have two polynomials with \( n \) coefficients (degree \( n - 1 \))

\[
\begin{array}{ccc}
  n-1 & n/2 & 0 \\
  P: & P_L & P_R \\
  Q: & Q_L & Q_R \\
\end{array}
\]

\[
P(x) \cdot Q(x) = P_L(x) \cdot Q_L(x) \cdot x^n +
\]

\[
(P_R(x) \cdot Q_L(x) + P_L(x) \cdot Q_R(x)) \cdot x^{n/2} +
\]

\[
P_R(x) \cdot Q_R(x)
\]

\[
T(n) = 4 \cdot T(n/2) + O(n) = O(n^2)
\]

\leftarrow\text{Shown later}

Product of complex numbers

• The product of two complex numbers requires four multiplications:

\[
(a + bi)(c + di) = ac - bd + (bc + ad)i
\]

• Carl Friedrich Gauss (1777-1855) noticed that it can be done with just three: \( ac, bd \) and \( (a + b)(c + d) \)

\[
bc + ad = (a + b)(c + d) - ac - bd
\]

• A similar observation applies for polynomial multiplication.

Product of polynomials with Gauss’s trick

\[
R_1 = P_L Q_L
\]

\[
R_2 = P_R Q_R
\]

\[
R_3 = (P_L + P_R)(Q_L + Q_R)
\]

\[
PQ = P_L Q_L x^n + (P_R Q_L + P_L Q_R) x^{n/2} + P_R Q_R
\]

\[
T(n) = 3T(n/2) + O(n)
\]

Polynomial multiplication: recursive step
**Pattern of recursive calls**

- **Branching factor:** 3

**Complexity analysis**

- The time spent at level \( k \) is
  \[
  3^k \cdot O \left( \frac{n}{2^k} \right) = \left( \frac{3}{2} \right)^k \cdot O(n)
  \]
- For \( k = 0 \), runtime is \( O(n) \).
- For \( k = \log_2 n \), runtime is \( O(3^{\log_2 n}) \), which is equal to \( O(n^{\log_2 3}) \).
- The runtime per level increases geometrically by a factor of 3/2 per level. The sum of any increasing geometric series is, within a constant factor, simply the last term of the series.
- Therefore, the complexity is \( O(n^{1.59}) \).

**Useful reminders**

- Sum of geometric series with ratio \( r \):
  \[
  S = a + ar + ar^2 + ar^3 + \ldots + ar^{n-1}
  \]
  \[
  S = a \frac{1-r^n}{1-r}
  \]
- Logarithms:
  \[
  \log_b n = \log_b a \cdot \log_a n
  \]
  \[
  a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a}
  \]

**A popular recursion tree**

- **Branching factor:** 2
- Example: efficient sorting algorithms.
- Algorithms may differ on the amount of work done at each level: \( O(n^c) \)
Examples

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Branch</th>
<th>c</th>
<th>Runtime equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power ($x^y$)</td>
<td>1</td>
<td>0</td>
<td>$T(y) = T(y/2) + O(1)$</td>
</tr>
<tr>
<td>Binary search</td>
<td>1</td>
<td>0</td>
<td>$T(n) = T(n/2) + O(1)$</td>
</tr>
<tr>
<td>Merge sort</td>
<td>2</td>
<td>1</td>
<td>$T(n) = 2 \cdot T(n/2) + O(n)$</td>
</tr>
<tr>
<td>Polynomial product</td>
<td>4</td>
<td>1</td>
<td>$T(n) = 4 \cdot T(n/2) + O(n)$</td>
</tr>
<tr>
<td>Polynomial product (Gauss)</td>
<td>3</td>
<td>1</td>
<td>$T(n) = 3 \cdot T(n/2) + O(n)$</td>
</tr>
</tbody>
</table>

Master theorem

- Typical pattern for Divide&Conquer algorithms:
  - Split the problem into $a$ subproblems of size $n/b$
  - Solve each subproblem recursively
  - Combine the answers in $O(n^c)$ time

- Running time: $T(n) = a \cdot T(n/b) + O(n^c)$

- Master theorem:

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

Master theorem: recursion tree

- For simplicity, assume $n$ is a power of $b$.
- The base case is reached after $\log_b n$ levels.
- The $k$th level of the tree has $a^k$ subproblems of size $n/b^k$.
- The total work done at level $k$ is:

$$
a^k \times 0 \left(\frac{n}{b^k}\right)^c = O(n^c) \times \left(\frac{a}{b^c}\right)^k
$$

- As $k$ goes from 0 (the root) to $\log_b n$ (the leaves), these numbers form a geometric series with ratio $a/b^c$. We need to find the sum of such a series.

$$
T(n) = O(n^c) \cdot \left(1 + \frac{a}{b^c} + \frac{a^2}{b^{2c}} + \frac{a^3}{b^{3c}} + \cdots + \frac{a^{\log_b n}}{b^{(\log_b n)c}}\right)
$$
Master theorem: proof

- Case $a/b^c < 1$. Decreasing series. The sum is dominated by the first term ($k = 0$): $O(n^c)$.

- Case $a/b^c > 1$. Increasing series. The sum is dominated by the last term ($k = \log_b n$):

$$n^c \left( \frac{a}{b} \right)^{\log_b n} = n^c \left( \frac{a^{\log_b n}}{(b^{\log_b n})^c} \right) = a^{\log_b n} = n^{\log_b a}$$

- Case $a/b^c = 1$. We have $O(\log n)$ terms all equal to $O(n^c)$.

Master theorem: examples

Running time: $T(n) = a \cdot T(n/b) + O(n^c)$

$$T(n) = \begin{cases} 
O(n^c) & \text{if } a < b^c \\
O(n^c \log n) & \text{if } a = b^c \\
O(n^{\log_b a}) & \text{if } a > b^c 
\end{cases}$$

Quick sort (Tony Hoare, 1959)

- Suppose that we know a number $x$ such that one-half of the elements of a vector are greater than or equal to $x$ and one-half of the elements are smaller than $x$.
  - Partition the vector into two equal parts ($n - 1$ comparisons)
  - Sort each part recursively

- Problem: we do not know $x$.

- The algorithm also works no matter which $x$ we pick for the partition. We call this number the pivot.

- Observation: the partition may be unbalanced.
Quick sort

The key step of quick sort is the partitioning algorithm.

**Question:** how to find a good pivot?

```plaintext
function Partition(A, left, right)
  // A[left..right]: segment to be sorted
  // Returns the middle of the partition with
  //   A[middle] = pivot
  //   A[left..middle-1] ≤ pivot
  //   A[middle+1..right] > pivot

  x = A[left]; // the pivot
  i = left; j = right;

  while i < j do
    while i ≤ right and A[i] ≤ x do i = i+1;
    while j ≥ left and A[j] > x do j = j-1;
    if i < j then swap(A[i], A[j]);

  swap(A[left], A[j]);
  return j;
```

Quick sort partition: example

function Qsort(A, left, right)
// A[left..right]: segment to be sorted
if left < right then
  mid = Partition(A, left, right);
  Qsort(A, left, mid-1);
  Qsort(A, mid+1, right);

Quick sort: algorithm

function HoarePartition(A, left, right)
// A[left..right]: segment to be sorted.
// Output: The left part has elements ≤ than the pivot.
// The right part has elements ≥ than the pivot.
// Returns the index of the last element of the left part.

x = A[left]; // the pivot
i = left-1; j = right+1;

while true do
  do i = i+1; while A[i] < x;
  do j = j-1; while A[j] > x;
  if i ≥ j then return j;
  swap(A[i], A[j]);

Admire a unique piece of art by Hoare:
The first swap creates two sentinels. After that, the algorithm flies ...

Quick sort partition: example
Quick sort with Hoare’s partition

```plaintext
function Qsort(A, left, right)
// A[left..right]: segment to be sorted
if left < right then
    mid = HoarePartition(A, left, right);
    Qsort(A, left, mid);
    Qsort(A, mid+1, right);

Divide & Conquer © Dept. CS, UPC
```

Quick sort: hybrid approach

```plaintext
function Qsort(A, left, right)
// A[left..right]: segment to be sorted.
// K is a break-even size in which insertion sort is
// more efficient than quick sort.
if right - left ≥ K then
    mid = HoarePartition(A, left, right);
    Qsort(A, left, mid);
    Qsort(A, mid+1, right);

function Sort(A):
    Qsort(A, 0, A.size()-1);
    InsertionSort(A);

Divide & Conquer © Dept. CS, UPC
```

Quick sort: complexity analysis

- The partition algorithm is $O(n)$.
- Assume that the partition is balanced:
  \[
  T(n) = 2 \cdot T(n/2) + O(n) = O(n \log n)
  \]
- Worst case runtime: the pivot is always the smallest element in the vector $\Rightarrow O(n^2)$
- Selecting a good pivot is essential. There are different strategies, e.g.,
  - Take the median of the first, last and middle elements
  - Take the pivot at random
- Let us assume that $x_i$ is the $i$th smallest element in the vector.
- Let us assume that each element has the same probability of being selected as pivot.
- The runtime if $x_i$ is selected as pivot is:
  \[
  T(n) = n - 1 + T(i - 1) + T(n - i)
  \]
Quick sort: complexity analysis

\[ T(n) = n - 1 + \frac{1}{n} \sum_{i=1}^{n} (T(i - 1) + T(n - i)) \]

\[ T(n) = n - 1 + \frac{1}{n} \sum_{i=1}^{n} T(i - 1) + \frac{1}{n} \sum_{i=1}^{n} T(n - i) \]

\[ T(n) = n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} T(i) \leq 2(n + 1)(H(n + 1) - 1.5) \]

\[ H(n) = 1 + 1/2 + 1/3 + \cdots + 1/n \] is the Harmonic series, that has a simple approximation: \[ H(n) = \ln n + \gamma + O(1/n). \]

\[ \gamma = 0.577 \ldots \] is Euler’s constant. [see the appendix]

Quick sort: complexity analysis summary

- Runtime of quicksort:
  \[ T(n) = O(n^2) \]
  \[ T(n) = \Omega(n \log n) \]
  \[ T_{avg}(n) = O(n \log n) \]

- Be careful: some malicious patterns may increase the probability of the worst case runtime, e.g., when the vector is sorted or almost sorted.

- Possible solution: use random pivots.

The selection problem

- Given a collection of \( N \) elements, find the \( k \)th smallest element.

- Options:
  - Sort a vector and select the \( k \)th location: \( O(N \log N) \)
  - Read \( k \) elements into a vector and sort them. The remaining elements are processed one by one and placed in the correct location (similar to insertion sort). Only \( k \) elements are maintained in the vector. Complexity: \( O(kN) \). Why?

The selection problem using a heap

- Algorithm:
  - Build a heap from the collection of elements: \( O(N) \)
  - Remove \( k \) elements: \( O(k \log N) \)
  - Note: heaps will be seen later in the course

- Complexity:
  - In general: \( O(N + k \log N) \)
  - For small values of \( k \), i.e., \( k = O(N/\log N) \), the complexity is \( O(N) \).
  - For large values of \( k \), the complexity is \( O(k \log N) \).
Quick sort with Hoare’s partition

function Qsort(A, left, right)
// A[left..right]: segment to be sorted
if left < right then
    mid = HoarePartition(A, left, right);
    Qsort(A, left, mid);
    Qsort(A, mid+1, right);

Quick select with Hoare’s partition

function Qselect(A, left, right, k)
if left == right then return A[left];
mid = HoarePartition(A, left, right);
// We only need to sort one half of A
if k ≤ mid then return Qselect(A, left, mid, k);
else return Qselect(A, mid+1, right, k);

Quick Select: complexity

• Assume that the partition is balanced:
  – Quick sort: \( T(n) = 2T(n/2) + O(n) = O(n \log n) \)
  – Quick select: \( T(n) = T(n/2) + O(n) = O(n) \)

• The average linear time complexity can be achieved by choosing good pivots (similar strategy and complexity computation to qsort).

The Closest-Points problem

• Input: A list of \( n \) points in the plane \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\)
• Output: The pair of closest points
• Simple approach: check all pairs \( \Rightarrow O(n^2) \)
• We want an \( O(n \log n) \) solution!
The Closest-Points problem

- We can assume that the points are sorted by the x-coordinate. Sorting the points is free from the complexity standpoint ($O(n \log n)$).

- Split the list into two halves. The closest points can be both at the left, both at the right or one at the left and the other at the right (center).

- The left and right pairs are easy to find (recursively). How about the pairs in the center?

Let us take all points in the strip and sort them by the y-coordinate. We only need to consider pairs of points with distance smaller than $\delta$.

Once we find a pair $(p_i, p_j)$ with y-coordinates that differ by more than $\delta$, we can move to the next $p_i$.

```plaintext
for (i=0; i < NumPointsInStrip; ++i)
    for (j=i+1; j < NumPointsInStrip; ++j)
        if (p_i and p_j's y-coordinate differ by more than $\delta$) break; // Go to next p_i
        if (dist(p_i, p_j) < $\delta$) $\delta = dist(p_i, p_j)$;
```

But, how many pairs $(p_i, p_j)$ do we need to consider?

- For every point $p_i$ at one side of the strip, we only need to consider points from $p_i+1$.

- The relevant points only reside in the $2\delta \times \delta$ rectangle below point $p_i$. There can only be 8 points at most in this rectangle (4 at the left and 4 at the right). Some points may have the same coordinates.
The Closest-Points problem: algorithm

• Sort the points according to their $x$-coordinates.

• Divide the set into two equal-sized parts.

• Compute the min distance at each part (recursively). Let $\delta$ be the minimal of the two minimal distances.

• Eliminate points that are farther than $\delta$ from the separation line.

• Sort the remaining points according to their $y$-coordinates.

• Scan the remaining points in the $y$ order and compute the distances of each point to its 7 neighbors.

• If any of these distances is smaller than $\delta$, update $\delta$.

The Closest-Points problem: complexity

• Initial sort using $x$-coordinates: $O(n \log n)$. It comes for free.

• Divide and conquer:
  – Solve for each part recursively: $2T(n/2)$
  – Eliminate points farther than $\delta$: $O(n)$
  – Sort remaining points using $y$-coordinates: $O(n \log n)$
  – Scan the remaining points in $y$ order: $O(n)$

$$T(n) = 2T(n/2) + O(n) + O(n \log n) = O(n \log^2 n)$$

• Can we do it in $O(n \log n)$? Yes, we need to sort by $y$ in a smart way.

Subtract and Conquer

• Sometimes we may find recurrences with the following structure:

$$T(n) = a \cdot T(n - b) + O(n^c)$$

• Examples:

  Hanoi($n$) = $2 \cdot$ Hanoi($n - 1$) + $O(1)$
  Sort($n$) = Sort($n - 1$) + $O(n)$

• Muster theorem:

$$T(n) = \begin{cases} 
0(n^c) & \text{if } a < 1 \quad \text{(never occurs)} \\
0(n^{c+1}) & \text{if } a = 1 \\
0(n^c a^{n/b}) & \text{if } a > 1 
\end{cases}$$
Muster theorem: recursion tree

\begin{align*}
\text{Size } n & \quad \text{Branching factor } a \\
\text{Size } n - b & \\
\text{Size } n - 2b & \\
\text{Size 1} & \quad \text{Width } a^{n/b} \quad \text{Depth } n/b
\end{align*}

Muster theorem: proof

- Expanding the recursion (assume that \( f(n) \) is \( O(n^c) \))

\[
T(n) = aT(n-b) + f(n)
\]

\[
= a(aT(n-2b) + f(n-b)) + f(n)
\]

\[
= a^2T(n-2b) + af(n-b) + f(n)
\]

\[
= a^3T(n-3b) + a^2f(n-2b) + af(n-b) + f(n)
\]

- Hence:

\[
T(n) = \sum_{i=0}^{\lfloor n/b \rfloor} a^i \cdot f(n-ib)
\]

- Since \( f(n-ib) \) is in \( O((n-ib)^c) \), which is in \( O(n^c) \), then

\[
T(n) = O \left( n^c \sum_{i=0}^{\lfloor n/b \rfloor} a^i \right)
\]

- The proof is completed by this property:

\[
\sum_{i=0}^{\lfloor n/b \rfloor} a^i = \begin{cases} 
0(1), & \text{if } a < 1 \\
0(n), & \text{if } a = 1 \\
0(a^{n/b}), & \text{if } a > 1
\end{cases}
\]

Muster theorem: examples

- **Hanoi:** \( T(n) = 2T(n-1) + O(1) \)

  We have \( a = 2 \) and \( c = 0 \), thus \( T(n) = O(2^n) \).

- **Selection sort** (recursive version):
  - Select the min element and move it to the first location
  - Sort the remaining elements

\[
T(n) = T(n-1) + O(n) \quad (a = c = 1)
\]

Thus, \( T(n) = O(n^2) \)

Fibonacci: \( T(n) = T(n-1) + T(n-2) + O(1) \)

We can compute bounds:

\[
2T(n-2) + O(1) \leq T(n) \leq 2T(n-1) + O(1)
\]

Thus, \( O(2^{n/2}) \leq T(n) \leq O(2^n) \)
The skyline problem

Given the exact locations and shapes of several rectangular buildings in a city, draw the skyline (in two dimensions) of these buildings, eliminating hidden lines (source: Udi Manber, Introduction to Algorithms, Addison-Wesley, 1989).

Input:

\[(1,1,5) (2,6,7) (3,13,9) (12,7,16) (14,3,25) (19,18,22) (23,13,29) (24,4,28)\]

Output:

\[(1,11,13,9,0,12,7,16,3,19,18,22,3,23,13,29,0)\]
(numbers in boldface represent heights)

Describe (in natural language) two different algorithms to solve the skyline problem:

• By induction: assume that you know how to solve it for \(n - 1\) buildings.
• Using Divide & Conquer: solve the problem for \(n/2\) buildings and combine.

Analyze the cost of each solution.

A, B or C?

Suppose you are choosing between the following three algorithms:

• Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.

• Algorithm B solves problems of size \(n\) by recursively solving two subproblems of size \(n - 1\) and then combining the solutions in constant time.

• Algorithm C solves problems of size \(n\) by dividing them into nine subproblems of size \(n/3\), recursively solving each subproblem, and then combining the solutions in \(O(n^2)\) time.

What are the running times of each of these algorithms (in big-O notation), and which one would you choose?


Crazy sorting

Let \(T[i..j]\) be a vector with \(n = j - i + 1\) elements. Consider the following sorting algorithm:

a) If \(n \leq 2\) the vector is easily sorted (constant time).

b) If \(n \geq 3\), divide the vector into three intervals \(T[i..k - 1]\), \(T[k..l]\) and \(T[l+1..j]\), where \(k = i + \lfloor n/3 \rfloor\) and \(l = j - \lfloor n/3 \rfloor\). The algorithm recursively sorts \(T[i..l]\), then it sorts \(T[k..j]\), and finally sorts \(T[i..l]\).

• Proof the correctness of the algorithm.
• Analyze the asymptotic complexity of the algorithm (give a recurrence of the runtime and solve it).
The majority element

A majority element in a vector, $A$, of size $n$ is an element that appears more than $n/2$ times (thus, there is at most one). For example, the vector $[3, 3, 4, 2, 4, 4, 2, 2]$ has a majority element (4), whereas the vector $[3, 3, 4, 2, 4, 4, 2, 2]$ does not. If there is no majority element, your program should indicate this. Here is a sketch of an algorithm to solve the problem:

First, a candidate majority element is found (this is the hardest part). This candidate is the only element that could possibly be the majority element. The second step determines if this candidate is actually the majority. This is just a sequential search through the vector. To find a candidate in the vector, $A$, form a second vector, $B$. Then compare $A_2$ and $A_1$. If they are equal, add one of these to $B$; otherwise do nothing. Then compare $A_3$ and $A_2$. Again if they are equal, add one of these to $B$; otherwise do nothing. Continue in this fashion until the entire vector is read. The recursively find a candidate for $B$; this is the candidate for $A$ (why?).

• How does the recursion terminate?
• What is the running time of the algorithm?
• How can we avoid using an extra array, $B$?
• Prove the correctness of the algorithm (hint: prove it for $n$ even)
• How is the case where $n$ is odd handled?


**APPENDIX**

Breaking into pieces

Let us assume that $f$ is $\Theta(1)$ and $g$ has a runtime proportional to the size of the vector it has to process, i.e., $\Theta(j - i + 1)$. What is the asymptotic cost of $A$ and $B$ as a function of $n$? ($n$ is the size of the vector).

If both functions do the same, which one would you choose?

```c
double A(vector<double>& v, int i, int j) {
    if (i < j) {
        int x = f(v, i, j);
        int m = (i+j)/2;
        return A(v, i, m-1) + A(v, m, j) + A(v, i+1, m) + x;
    } else {
        return v[i];
    }
}

double B(vector<double>& v, int i, int j) {
    if (i < j) {
        int x = g(v, i, j);
        int m1 = i + (j-i+1)/3;
        int m2 = i + (j-i+1)*2/3;
        return B(v, i, m1-1) + B(v, m1, m2-1) + B(v, m2, j) + x;
    } else {
        return v[i];
    }
}
```

Logarithmic identities

<table>
<thead>
<tr>
<th>Identity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b^{\log_b a} = \log_b b^a = a$</td>
</tr>
<tr>
<td>$\log_b (xy) = \log_b x + \log_b y$</td>
</tr>
<tr>
<td>$\frac{x}{y} = \log_b x - \log_b y$</td>
</tr>
<tr>
<td>$\log_b x^c = c \log_b x$</td>
</tr>
<tr>
<td>$\log_b x = \frac{\log_c x}{\log_c b}$</td>
</tr>
<tr>
<td>$x^{\log_b y} = y^{\log_b x}$</td>
</tr>
</tbody>
</table>

\[
\gamma = \lim_{n \to \infty} \left( -\ln n + \sum_{k=1}^{n} \frac{1}{k} \right) \quad \rightarrow \quad \sum_{k=1}^{n} \frac{1}{k} \in \Theta(\log n)
\]

\[
\gamma = 0.5772 \ldots \text{ (Euler-Mascheroni constant)}
\]
A full-history recurrence relation

\[ T(n) = n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} T(i) \]

A recurrence that depends on all the previous values of the function.

\[ nT(n) = n(n-1) + 2 \sum_{i=0}^{n-1} T(i), \quad (n + 1)T(n + 1) = (n + 1)n + 2 \sum_{i=0}^{n} T(i) \]

\[ (n + 1)T(n + 1) - nT(n) = (n + 1)n - n(n - 1) + 2T(n) = 2n + 2T(n) \]

\[ T(n + 1) = \frac{n + 2}{n + 1} T(n) + \frac{2n}{n + 1} \leq \frac{n + 2}{n + 1} T(n) + 2 \]

\[ T(n) \leq 2 + \frac{n + 1}{n} \left( 2 + \frac{n}{n - 1} \left( 2 + \frac{n - 1}{n - 2} \left( \ldots + \frac{4}{3} \right) \right) \right) \]

\[ T(n) \leq 2 \left( 1 + \frac{n + 1}{n} + \frac{n + 1}{n - 1} + \frac{n + 1}{n - 2} + \ldots + \frac{1}{3} \right) = 2(n + 1) \left( \frac{1}{n + 1} + \frac{1}{n} + \frac{1}{n - 1} + \ldots + \frac{1}{3} \right) \]

\[ T(n) \leq 2(n + 1)(H(n + 1) - 1.5) = \Theta(n \log n) \]