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.
The key step of quick sort is the partitioning algorithm.

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

**Quick sort: partition**

```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**
**Quick sort: algorithm**

**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);

Divide & Conquer © Dept. CS, UPC

**Quick sort partition: example**

4 2 3 5 10 9 12 1 15 13 6 11 16 14

**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);

Divide & Conquer © Dept. CS, UPC
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);

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

$H(n) = 1 + 1/2 + 1/3 + \cdots + 1/n$ 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]

$$T(n) \leq 2(n + 1)(\ln n + \gamma - 1.5) + O(1) \in O(n \log n)$$
Quick sort: complexity analysis summary

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

- 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?

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);
```

Quick select with Hoare’s partition

```plaintext
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 \leq 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: \( T(n) = 2T(n/2) + \mathcal{O}(n) = \mathcal{O}(n \log n) \)
  - Quick select: \( T(n) = T(n/2) + \mathcal{O}(n) = \mathcal{O}(n) \)

- 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 \( \rightarrow \mathcal{O}(n^2) \)
- We want an \( \mathcal{O}(n \log n) \) solution!

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

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 \( \delta = \min(\delta_L, \delta_R) \). We only need to compute \( \delta_C \) if it improves \( \delta \).

We can define a strip around the center with distance \( \delta \) at the left and right. If \( \delta_C \) improves \( \delta \), then the points must be within the strip.

In the worst case, all points can still reside in the strip.

But how many points do we really have to consider?
The Closest-Points problem

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?

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.

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.
The Closest-Points problem: complexity

• Let \( Y \) a vector with the points sorted by the \( y \)-coordinates. This can be done initially for free.

• Each time we partition the set of points by the \( x \)-coordinate, we also partition \( Y \) into two sorted vectors (using an “unmerging” procedure with linear complexity)

\[
Y_L = Y_R = \emptyset \quad // \text{Initial lists of points}
\]

foreach \( p_i \in Y \) in ascending order of \( y \) do
  if \( p_i \) is at the left part then \( Y_L \).push_back(\( p_i \))
  else \( Y_R \).push_back(\( p_i \))

• Now, sorting the points by the \( y \)-coordinate at each iteration can be done in linear time, and the problem can be solved in \( O(n \log n) \)

Subtract and Conquer

• Sometimes we may find recurrences with the following structure:

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

• Examples:

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

  Muster theorem:

\[
T(n) = \begin{cases} 
  O(n^c) & \text{if } a < 1 \quad (\text{never occurs}) \\
  O(n^{c+1}) & \text{if } a = 1 \\
  O(n^c a^{n/b}) & \text{if } a > 1
\end{cases}
\]

Muster theorem: recursion tree

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, \( T(n) = O(n^2) \)
Muster theorem: examples

**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,

\[
O\left(2^{n/2}\right) \leq T(n) \leq O(2^n)
\]

EXERCICES

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 

**Input:**

\[(1, 11, 5) (2, 6, 7) (3, 13, 9) (12, 7, 16) (14, 3, 25) (19, 18, 22) (23, 13, 29) (24, 4, 28)\]

**Output:**

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

(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 $k = i + \lceil n/3 \rceil$ 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).

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?

```cpp
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];
    }
}
```

APPENDIX

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 $[3,3,4,2,4,4,2,4,4]$ 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 $A_3$. 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?

Logarithmic identities

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

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

$\gamma = 0.5772 \ldots$ (Euler-Mascheroni constant)

Harmonic series

Full-history recurrence relation

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

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

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

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

$$T(n) \leq 2 + \frac{n+1}{n} \left( 2 + \frac{n}{n-1} \left( 2 + \frac{n-1}{n-2} \left( \cdots + \frac{4}{3} \right) \right) \right)$$

$T(n) \leq 2 \left( 1 + \frac{n+1}{n} + \frac{n+1}{n-1} + \frac{n+1}{n-2} + \cdots + \frac{n+1}{n-n-1} \right)$

$T(n) \leq 2(n+1) \left( \frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{3} \right)$

$$T(n) \leq 2(n+1)(H(n+1) - 1.5) \in \Theta(n \log n)$$

Muster theorem: proof

- Expanding the recursion (assume that $f(n)$ is $O(n^c)$)
  $$T(n) = aT(n-b) + f(n)$$
  $$= a(aT(n-2b) + f(n-b)) + f(n)$$
  $$= a^2T(n-3b) + af(n-b) + f(n)$$
  $$= a^2T(n-3b) + a^2f(n-2b) + af(n-b) + f(n)$$
- Hence:
  $$T(n) = \sum_{i=0}^{n/b} a^i \cdot f(n-i\cdot b)$$
- Since $f(n-i\cdot b)$ is in $O((n-i\cdot b)^c)$, which is in $O(n^c)$, then
  $$T(n) = O\left( n^c \sum_{i=0}^{n/b} a^i \right)$$
- The proof is completed by this property:
  $$\sum_{i=0}^{n/b} a^i = \begin{cases} 
  0(1), & \text{if } a < 1 \\
  0(n), & \text{if } a = 1 \\
  0(a^{n/b}), & \text{if } a > 1 
  \end{cases}$$