There are \( m \) nuts of distinct sizes \((n_1, \ldots, n_m)\) and \( m \) bolts of distinct sizes \((b_1, \ldots, b_m)\). Each bolt fits exactly one nut. You may not compare two nuts directly to each other, nor may you compare two bolts directly to each other. The only (benchmark) operation you may do is to compare a nut, say \( n_i \), to a bolt, say \( b_j \). The three possible outcomes of a comparison are that
- \( n_i < b_j \), the nut is too small for the bolt,
- \( n_i = b_j \), the nut fits the bolt,
- \( n_i > b_j \), the nut is too large for the bolt.

Find an algorithm for finding the correct nut for each bolt which executes in expected time in \( \Theta(m \log m) \).

**SOLUTION:**

```python
NUTBOLTMATCH(nuts, bolts)
if (|nuts|>1 V |bolts|>1) then
    LargeNuts, SmallNuts, LargeBolts, SmallBolts <- Ø
Select and remove a random nut \( n^* \) from nuts
for each bolt \( b \)
    if \( b = n^* \) then call the bolt \( b^* \)
    else if \( b > n^* \) then LargeBolts <- LargeBolts \cup \{b\}
    else SmallBolts <- SmallBolts \cup \{b\}
for each nut \( n \)
    if \( b^* > n \) then SmallNuts <- SmallNuts \cup \{n\}
    else LargeNuts <- LargeNuts \cup \{n\}
NUTBOLTMATCH(SmallNuts, SmallBolts)
NUTBOLTMATCH(LargeNuts, LargeBolts)
return \((n^*, b^*)\)
```

Since the algorithm is essentially QUICKSORTing \( nuts \) and \( bolts \), the expected time of the two QUICKSORTs is \( \Theta(m \log m) \).

---

Suppose you have an algorithm \( A_1 \) to solve a problem of size \( n \) in \( \Theta(n^2) \) steps. You discover a divide-&-conquer algorithm \( A_2 \) which can, in \( \Theta(n \log n) \) time, divide a problem of size \( n \) into two subproblems of size \( n/2 \), and then, in time \( \Theta(n \log n) \), recombine two solutions of size \( n/2 \) into a solution of the original problem. Which algorithm, \( A_1 \) or \( A_2 \), is more efficient? You may assume that \( n \) is a power of 2.

**SOLUTION:** Letting \( T(n) \) denote the time for \( A_2 \) to solve the problem, we derive the recurrence

\[
T(n) = 2T(n/2) + cn \log n
\]

where \( c \) is the sum of the constants hidden in the asymptotic notation for the complexities of the division and the recombination of \( A_2 \). Unfolding the recurrence,

\[
T(n) = 2T(n/2) + cn \log n
\]

\[
= 2 \left( 2T(n/4) + cn \frac{n}{2} \right) + cn \log n = 4T(n/4) + cn \log n + cn(\log n - 1)
\]
After $k$ levels of unfolding,

$$T(n) = 2^k T(n/2^k) + cn\left(\left\lfloor \frac{\log^2 n - \log n (\log n - 1)}{2} \right\rfloor \right)$$

The unfolding stops when $2^k = n$, or $k = \log n$. So

$$T(n) = 2^k T(n/2^k) + cn\left(\left\lfloor \frac{\log^2 n - \log n (\log n - 1)}{2} \right\rfloor \right) = n + cn\left(\left\lfloor \frac{\log^2 n - \log n (\log n - 1)}{2} \right\rfloor \right) \in \Theta(n \log^2 n)$$

3 Halving is the process of dividing a set $S$ into essentially equal sized subsets, $S_1, S_2$, $|S_1| = |S_2| \pm 1$ such that $(\forall x \in S_1)(\forall y \in S_2)x \leq y$. Assume the median of a set of $n$ elements can be found using at most $t_m(n)$ pairwise comparisons, and a set of $n$ elements can be halved using at most $t_h(n)$ pairwise comparisons.

**a** Show that $t_h(n) \leq t_m(n) + n - 1$.

**b** Show that $t_m(n) \leq t_h(n) + \left\lceil n/2 \right\rceil - 1$.

**SOLUTION:** Without loss of generality, assume that $|S_1| \geq |S_2|$.

**a** compute the median $\mu$ of $S$

$$S \leftarrow S \setminus \{\mu\}$$

$S_1, S_2 \leftarrow \emptyset$

**for each** $s \in S$ **do**

- **if** $s \leq \mu$ **then** $S_1 \leftarrow S_1 \cup \{s\}$ **else** $S_2 \leftarrow S_2 \cup \{s\}$

$n-1$

$S_1 \leftarrow S_1 \cup \{\mu\}$

**if** $|S_1| > |S_2| + 1$ **then** move enough $\mu$s from $S_1$ to $S_2$ to balance them

**b** halve $S$

$$t_h(n)$$

**return** max($S_1$)

- \left\lceil n/2 \right\rceil - 1


**SOLUTION:**

```
MERGE SORT(A)
for i \leftarrow 1 \ to \ n - 1 \ do
     \Theta(n \log n)
```

```
for i \leftarrow 1 \ to \ n - 1 \ do
     O(n)
```
if $A[i] = A[i+1]$ then return "All elements are not distinct."

return "All elements are distinct." $O(1)$

To show that $\Omega(n \log n)$ comparisons are necessary in the worst-case, we assume that all the elements are distinct. We note that it takes $\Omega(n^2)$ comparisons to compare each pair of elements. So assume we haven’t compared each pair of elements. The only way any algorithm can infer $A[i] \neq A[j]$ for some pair $(i, j)$ is to have performed some nonempty sequence of comparisons yielding the results $A[i] < A[k_1], A[k_1] < A[k_2], \ldots, A[k_l] < A[j]$. Hence the algorithm must "know" (must have access to information permitting the inference) the relative order between every pair of elements. Since this is enough information to permit sorting $A$, the $\Omega(n \log n)$ bound on the number of pairwise comparisons needed to sort $n$ numbers must apply.

5 Consider the following divide-and-conquer algorithm for computing a minimum spanning tree.

INPUT: Connected weighted graph $G = (V, E)$ with $w : E \rightarrow \mathbb{R}^+$. OUTPUT: The edges of a minimum spanning tree of $G$

$MST(V, E)$

if $|V| = 1$ then return $\emptyset$

if $|V| = 2$ then return $E$

Partition $V$ into $V_1$ and $V_2$ such that $\|V_1| - |V_2\| \leq 1$

Let $E_1$ be the edges incident only with vertices of $V_1$

Let $E_2$ be the edges incident only with vertices of $V_2$

Let $e$ be a lightest edge incident with a vertex of $V_1$ and a vertex of $V_2$

return $MST(V_1, E_1) \cup MST(V_2, E_2) \cup \{e\}$

Does MST always produce a minimum spanning tree? Justify your answer.

SOLUTION: The algorithm does not always work. Consider the graph

If the partition is $\{V_1, V_2\} = \{\{a, b\}, \{c, d\}\}$, then the algorithm will return the set of edges $\{ab, ac, cd\}$ of weight 62. The minimum spanning tree is $\{ac, bd, ab\}$ of weight 33.

6 A majority element of an array is an element that appears in $>n/2$ positions. Give a linear time divide-&-conquer algorithm to test if an array has a majority element.
**SOLUTION:** Assume $n$ is even.

$CurrentMajority$ is undefined, with an excess of 0

for each of the $n/2$ pairs $(a_i, a_{i+1})$

- if $a_i = a_j = CurrentMajority$ then increase excess of $CurrentMajority$ by 2
- if $a_i = a_j \land a_i \neq CurrentMajority$ and $CurrentMajority$ defined then reduce excess of $CurrentMajority$ by 2
- if $a_i = a_j \land a_i \neq CurrentMajority \land CurrentMajority$ not defined then $CurrentMajority \leftarrow a_i$ with an excess of 2
- if excess of $CurrentMajority = 0$ then $CurrentMajority$ is undefined

Check if $CurrentMajority$ is a majority element in another linear time

Prove that if the array has a majority element before the above operation then it has a majority element after the operation.