D0012E Lab2

2025-12-04

Gustav Merelius
gusmer-4@student.ltu.se

Irma Palo Hjärtström
irmpal-1@student.ltu.se

John Kågström
johkgn-2@student.ltu.se

Contents

1 Algorithm 1: Three Sum

Given an array A of $n(n \geq 4)$ distinct real numbers, the problem is to determine whether or not A contains three distinct elements $x, y,$ and z such that $x + y = z$. You may assume that $n = 2^k$ for some positive integer $k \geq 2$.

- Design a worst-case $\Theta(n^3)$-time recursive algorithm using an incremental approach to solve the above problem.

For $\Theta(n^3)$-time there is only one function that takes the array, x, y and z as arguments and the function is called recursively, looping through the array three times as Array[x], Array[y] and Array[z]. x is the "outer"-most loop and z the "inner"-most loop.

1.1 Step-by-step

1. Check if x has reached the end, this means that there is no solution for the given problem.
2. Check if y has reached the end, this means that every combination for the current index of x has been checked, and the function is called recursively with the next value for x.
3. Check if z has reached the end, this means that every combination for the current index of y has been checked, and the function is called recursively with the next value for y.
4. If $x \neq y, z \neq y$ and $z \neq x$ and we find a combination so that array[x] + array[y] = array[z] the loop breaks and returns true.
5. If non of the previous statements are fulfilled the function must check the next value for the inner most loop, meaning the function is called again with the next value for z.

- Design a worst-case $\Theta(n^2)$-time recursive algorithm using an incremental approach to solve the above problem.

This algorithm first sorts the array using incremental recursive insertion sort. After sorting it uses the two-pointer method where x is set to 0 and y to z-1. Since the array is sorted we can check if x+y is smaller or larger than z, if that is the case move the corresponding pointer either up or down.

The length of the array is n. The index of the element to insert is l. r is a boolean used as a "sorting flag" - used to know when the sorting phase is done.

1.2 Step-by-step

1. Initial call: check that all parameters are `None` and r is `False`, also check that the array is $n > 3$. Call sorting with l=1 and r=True.
2. Sorting phase - starting when r is passed as True.
   1. Check if l has reached the end, if so: sorting is complete and the function is recursively called with r passed as true. Otherwise continue.
   2. The item on index Array[l] is stored in `key`. $j = l - 1$.
   3. Larger items in Array[$0 \rightarrow (l-1)$] are shifted right.
   4. Recursively call the function with the next value for l until done
   5. When done: r=False $\Rightarrow$ Search phase is initialized.
3. Search phase - starting when r is passed as False.
   1. Initialize z and y. Check whether z has reach the end or not, if so: searching is completed and there was no correct solutions. Otherwise continue.

2. Initialize x. Check whether x is larger or the same number as y, if that is the case it means the pointers overlap and no solution exists for that z. Call function recursively with the next z.
3. Compare target = Array[x] + Array[y] with Array[z]:
   - If target == Array[z] ⇒ we found a solution.
   - If target < Array[z] ⇒ move the x pointer up one step.
   - If target > Array[z] ⇒ move the y pointer down one step.

2 Algorithm 2: Divide-and-Conquer Maximum Subarray

Given an array $\boldsymbol{A} = \langle \boldsymbol{a_1}, \boldsymbol{a_2}, ..., \boldsymbol{a_n} \rangle$ where $\boldsymbol{a_i} \in \mathbb{R} \setminus \{\boldsymbol{0}\}$, the task is to find a *consecutive subarray* $\langle \boldsymbol{a_i}, \boldsymbol{a_{i+1}}, ..., \boldsymbol{a_j} \rangle$ whose *sum is maximum* among all possible consecutive subarrays. Using the *divide-and-conquer* method.

2.1 Description

The algorithm recursively splits the array into two halves, solves each half, and then merges the results to get the answer for the full interval.

Split the array, solve each half, then combine the partial results.

For every interval `[l, r)`, the algorithm computes a 4-tuple:

- `total` - sum of all elements in the interval
- `prefix` - maximum subarray sum that starts at index `l`
- `suffix` - maximum subarray sum that ends at index `r-1`
- `best` - maximum subarray sum anywhere inside the interval

2.1.1 Step-by-step

1. **Base case:** If the interval length is 1, containing a single value `val = nums[l]`, then `(total, prefix, suffix, best) = (val, val, val, val)`.
2. **Divide:** Split the interval at `mid = (l + r) // 2`.
   Recursively compute tuples for `[l, mid)` and `[mid, r)`.
3. **Conquer / Merge:** Using the tuples from left and right:
   - `total  = left.total + right.total`
   - `prefix = max(left.prefix, left.total + right.prefix)`
   - `suffix = max(left.suffix + right.total, right.suffix)`
   - `best   = max(left.suffix + right.prefix, left.best, right.best)`
4. **Final answer:** Call the function on `[0, n)`, and return the `best` field.

3 Algorithm 3: Sort with recursive fraction

3.1 Description

The **sortR** algorithm is a recursive comparison-based sorting algorithm using divide-and-conquer with overlapping subproblems (as introduced in Lec 06 on Divide & Conquer).

Given an array $\boldsymbol{A = \langle a_1, a_2, ..., a_n \rangle}$, the algorithm works as follows:

3.1.1 Step-by-step explanation

Step 0 (Base case):
If $n \leq 4$, sort the array using insertion sort and return.

Step 1:
Recursively sort the first $3n/4$ elements:
$b = \text{sortR}\big(\langle a_1, a_2, ..., a_{\lfloor 3n/4 \rfloor} \rangle\big)$

Step 2:
Recursively sort elements from position $\lfloor n/4 \rfloor + 1$ to n:
This combines $b_{\lfloor n/4 \rfloor + 1}, ..., b_{\lfloor 3n/4 \rfloor}$ with $a_{\lfloor 3n/4 \rfloor + 1}, ..., a_n$
$c = \text{sortR}(\text{combined elements})$

Step 3:
Recursively sort elements from position 1 to $\lfloor 3n/4 \rfloor$:
This combines $b_1, ..., b_{\lfloor n/4 \rfloor}$ with $c_{\lfloor n/4 \rfloor + 1}, ..., c_{\lfloor 3n/4 \rfloor}$
$d = \text{sortR}(\text{combined elements})$

Step 4 (Combine):
Return $\langle d_1, d_2, ..., d_{\lfloor 3n/4 \rfloor}, c_{\lfloor 3n/4 \rfloor + 1}, ..., c_n \rangle$

The algorithm uses three overlapping recursive calls on subarrays of size $3n/4$, ensuring that elements are repositioned correctly through multiple passes.

3.2 Correctness Proof using induction

We prove correctness by **strong induction** on n, following the proof technique from Lec 04 (Analyses).

3.2.1 Base Case

For $n \leq 4$, the algorithm uses insertion sort.

Insertion sort is a correct sorting algorithm (covered in Lec 05-Incremental), therefore **sortR** correctly sorts all arrays of size $n \leq 4$.

3.2.2 Induction Hypothesis

Assume that **sortR** correctly sorts all arrays of size m where $m < n$ for some $n > 4$.

3.2.3 Induction Step

Consider an array A of size $n > 4$. We must show that **sortR**(A) produces a correctly sorted array.

After Step 1:

The first $\lfloor 3n/4 \rfloor$ elements are sorted by the induction hypothesis (since $3n/4 < n$).

Result: $b = \langle b_1, b_2, ..., b_{\lfloor 3n/4 \rfloor} \rangle$ where $b_1 \leq b_2 \leq ... \leq b_{\lfloor 3n/4 \rfloor}$

After Step 2:

Elements from position $\lfloor n/4 \rfloor + 1$ to n are sorted by the induction hypothesis.

This merges already-sorted elements $b_{\lfloor n/4 \rfloor+1}, ..., b_{\lfloor 3n/4 \rfloor}$ with original elements $a_{\lfloor 3n/4 \rfloor+1}, ..., a_n$.

Result: $c = \langle c_{\lfloor n/4 \rfloor+1}, ..., c_n \rangle$ is sorted.

After Step 3:

Elements from position 1 to $\lfloor 3n/4 \rfloor$ are sorted by the induction hypothesis.

This merges $b_1, ..., b_{\lfloor n/4 \rfloor}$ with the refined $c_{\lfloor n/4 \rfloor+1}, ..., c_{\lfloor 3n/4 \rfloor}$ from Step 2.

Result: $d = \langle d_1, ..., d_{\lfloor 3n/4 \rfloor} \rangle$ is sorted.

Final result:

The output is $\langle d_1, ..., d_{\lfloor 3n/4 \rfloor}, c_{\lfloor 3n/4 \rfloor+1}, ..., c_n \rangle$.

Key properties:

- $d_1 \leq d_2 \leq ... \leq d_{\lfloor 3n/4 \rfloor}$ (sorted from Step 3)
- $c_{\lfloor 3n/4 \rfloor+1} \leq ... \leq c_n$ (sorted from Step 2)
- Step 3 ensures that d contains the smallest $\lfloor 3n/4 \rfloor$ elements of A in sorted order
- Step 2 ensures that the tail of c contains the largest $\lfloor n/4 \rfloor$ elements of A in sorted order
- Therefore: $d_{\lfloor 3n/4 \rfloor} \leq c_{\lfloor 3n/4 \rfloor+1}$

Thus, the concatenation produces a fully sorted array.

By the principle of strong induction, **sortR** correctly sorts arrays of all sizes $n \geq 1$.

3.3 Worst Case scenario recurrence equation

Let $T(n)$ be the worst-case running time of **sortR** on an input of size n.

Base case:

For $n \leq 4$, we use insertion sort:

$T(n) = \Theta(n^2)$ for $n \leq 4$

Since $n \leq 4$ is constant, this is $\Theta(1)$ in the analysis.

Recursive case ($n > 4$):

The algorithm performs:

1. Step 1: sort first $3n/4$ elements $\longrightarrow T(\lfloor 3n/4 \rfloor)$
2. Step 2: sort $3n/4$ elements $\longrightarrow T(\lfloor 3n/4 \rfloor)$
3. Step 3: sort $3n/4$ elements $\longrightarrow T(\lfloor 3n/4 \rfloor)$
4. Step 4: combine arrays (copying elements) $\longrightarrow \Theta(n)$

Therefore, the recurrence equation is:

$$T(n) = 3T(\lfloor 3n/4 \rfloor) + \Theta(n) \tag{1}$$

For analysis, we can ignore the floor function and write:

$$T(n) = 3T(3n/4) + \Theta(n) \tag{2}$$

3.4 Theorem 4.1 Master theorem

We solve the recurrence using the **Master Theorem** (Theorem 4.1 from Lec 06-Divide and Conquer).

The recurrence $T(n) = 3T(3n/4) + \Theta(n)$ is in the standard form:

$$T(n) = aT(n/b) + f(n) \tag{3}$$

where:

- $a = 3$ (three recursive subproblems)
- $b = 4/3$ (each subproblem is $3n/4 = n/(4/3)$ of the original size)
- $f(n) = \Theta(n)$ (work to combine results)

Step 1: Compute $n^{\log_b a}$:

$$\log_b a = \log_{4/3} 3 = \frac{\log 3}{\log(4/3)} = \frac{\log 3}{\log 4 - \log 3} \tag{4}$$

Numerically:

$$\log_{4/3} 3 \approx \frac{1.0986}{0.2877} \approx 3.82 \tag{5}$$

Therefore:

$$n^{\log_b a} = n^{\log_{4/3} 3} \approx n^{3.82} \tag{6}$$

Step 2: Compare $f(n)$ with $n^{\log_b a}$:

We have:

- $f(n) = \Theta(n) = \Theta(n^1)$
- $n^{\log_b a} = n^{3.82}$

Since n^1 grows **polynomially slower** than $n^{3.82}$, we can write:

$$f(n) = O\big(n^{\log_b a - \varepsilon}\big) \tag{7}$$

for $\varepsilon = 3.82 - 1 = 2.82 > 0$.

Step 3: Apply Master Theorem Case 1:

According to the Master Theorem (Lec 06), when $f(n) = O\big(n^{\log_b a - \varepsilon}\big)$ for some constant $\varepsilon > 0$, we have:

$$T(n) = \Theta\big(n^{\log_b a}\big) \tag{8}$$

Final result:

$$T(n) = \Theta\big(n^{\log_{4/3} 3}\big) = \Theta(n^{3.82}) \tag{9}$$

3.4.1 Conclusion

The worst-case running time of **sortR** is $\Theta(n^{3.82})$, which is significantly slower than optimal comparison-based sorting algorithms like merge sort that run in $\Theta(n \log n)$ time.

The inefficiency arises from the three overlapping recursive calls that reprocess portions of the array multiple times.