Algorithms and Data Structures

A7. Runtime Analysis: Bottom-Up Merge Sort

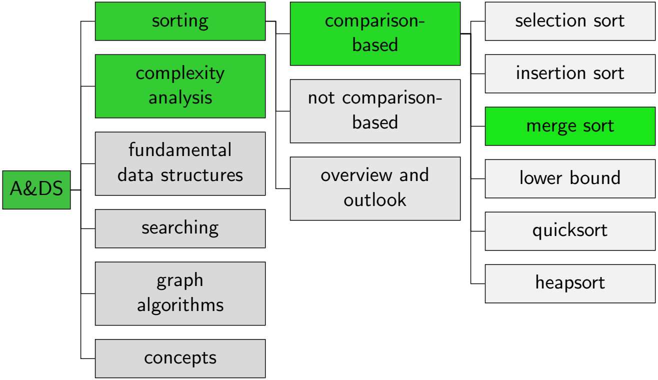
Gabriele Röger

University of Basel

March 7, 2024

Runtime Analysis: Bottom-Up Merge Sort

Content of the Course



Merge Step

```python
def merge(array, tmp, lo, mid, hi):
    i = lo
    j = mid + 1
    for k in range(lo, hi + 1):  # k = lo,...,hi
        if j > hi or (i <= mid and array[i] <= array[j]):
            tmp[k] = array[i]
            i += 1
        else:
            tmp[k] = array[j]
            j += 1
    for k in range(lo, hi + 1):  # k = lo,...,hi
        array[k] = tmp[k]
```

We analyze the running time for $m := \text{hi} - \text{lo} + 1$
(number of elements that should be merged).

Merge Step

```
     def merge(array, tmp, lo, mid, hi):
   |     i = lo
c1 |     j = mid + 1
         for k in range(lo, hi + 1):   # k = lo,...,hi
   |         if j > hi or (i <= mid and array[i] <= array[j]):
   |             tmp[k] = array[i]
c2 |             i += 1
   |         else:
   |             tmp[k] = array[j]
   |             j += 1
         for k in range(lo, hi + 1):   # k = lo,...,hi
c3 |         array[k] = tmp[k]
```

We analyze the running time for $m := \text{hi} - \text{lo} + 1$
(number of elements that should be merged).

Merge Step: Analysis

$$
\begin{aligned}
T(m) &= c_1 + c_2 m + c_3 m \\
&\geq (c_2 + c_3) m
\end{aligned}
$$

Merge Step: Analysis

$$\begin{aligned} T(m) &= c_1 + c_2 m + c_3 m \\ &\geq (c_2 + c_3) m \end{aligned}$$

For $m \geq 1$:

$$\begin{aligned} T(m) &= c_1 + c_2 m + c_3 m \\ &\leq c_1 m + c_2 m + c_3 m \\ &= (c_1 + c_2 + c_3) m \end{aligned}$$

Merge Step: Analysis

$$\begin{aligned} T(m) &= c_1 + c_2 m + c_3 m \\ &\geq (c_2 + c_3) m \end{aligned}$$

For $m \geq 1$:

$$\begin{aligned} T(m) &= c_1 + c_2 m + c_3 m \\ &\leq c_1 m + c_2 m + c_3 m \\ &= (c_1 + c_2 + c_3) m \end{aligned}$$

Theorem

The merge step has linear running time, i.e., there are constants $c, c', n_0 > 0$ such that for all $n \geq n_0$: $cn \leq T(n) \leq c'n$.

Bottom-Up Merge Sort

```
1  def sort(array):
2      n = len(array)
3      tmp = list(array)
4      length = 1
5      while length < n:
6          lo = 0
7          while lo < n - length:
8              mid = lo + length - 1
9              hi = min(lo + 2 * length - 1, n - 1)
10             merge(array, tmp, lo, mid, hi)
11             lo += 2 * length
12         length *= 2
```

We use the following constants in the analysis:

- c_1 lines 2–4
- c_2 lines 6 and 12
- c_3 lines 8,9,11

Assumption: merge requires c_4(hi-lo+1) operations.

Bottom-Up Merge Sort: Analysis I

Assumption: $n = 2^k$ for some $k \in \mathbb{N}_{>0}$

Bottom-Up Merge Sort: Analysis I

Assumption: $n = 2^k$ for some $k \in \mathbb{N}_{>0}$

Iterations of the outer loop (m for hi-lo+1):

Bottom-Up Merge Sort: Analysis I

Assumption: $n = 2^k$ for some $k \in \mathbb{N}_{>0}$

Iterations of the outer loop (m for hi-lo+1):

- Iteration 1: $n/2$ times inner loop with merge for $m = 2$
  $c_2 + n/2(c_3 + 2c_4) = c_2 + 0.5c_3 n + c_4 n$

Bottom-Up Merge Sort: Analysis I

Assumption: $n = 2^k$ for some $k \in \mathbb{N}_{>0}$

Iterations of the outer loop (m for hi-lo+1):

- Iteration 1: $n/2$ times inner loop with merge for $m = 2$
  $c_2 + n/2(c_3 + 2c_4) = c_2 + 0.5c_3 n + c_4 n$
- Iteration 2: $n/4$ times inner loop with merge for $m = 4$
  $c_2 + n/4(c_3 + 4c_4) = c_2 + 0.25c_3 n + c_4 n$

Bottom-Up Merge Sort: Analysis I

Assumption: $n = 2^k$ for some $k \in \mathbb{N}_{>0}$

Iterations of the outer loop (m for hi-lo+1):

- Iteration 1: $n/2$ times inner loop with merge for $m = 2$
  $c_2 + n/2(c_3 + 2c_4) = c_2 + 0.5c_3 n + c_4 n$
- Iteration 2: $n/4$ times inner loop with merge for $m = 4$
  $c_2 + n/4(c_3 + 4c_4) = c_2 + 0.25c_3 n + c_4 n$
- ...

Bottom-Up Merge Sort: Analysis I

Assumption: $n = 2^k$ for some $k \in \mathbb{N}_{>0}$

Iterations of the outer loop (m for hi-lo+1):

- Iteration 1: $n/2$ times inner loop with merge for $m = 2$
  $c_2 + n/2(c_3 + 2c_4) = c_2 + 0.5c_3 n + c_4 n$
- Iteration 2: $n/4$ times inner loop with merge for $m = 4$
  $c_2 + n/4(c_3 + 4c_4) = c_2 + 0.25c_3 n + c_4 n$
- ...
- Outer loop terminates after last iteration ℓ.

Bottom-Up Merge Sort: Analysis I

Assumption: $n = 2^k$ for some $k \in \mathbb{N}_{>0}$

Iterations of the outer loop (m for hi-lo+1):

- Iteration 1: $n/2$ times inner loop with merge for $m = 2$
  $c_2 + n/2(c_3 + 2c_4) = c_2 + 0.5c_3 n + c_4 n$
- Iteration 2: $n/4$ times inner loop with merge for $m = 4$
  $c_2 + n/4(c_3 + 4c_4) = c_2 + 0.25c_3 n + c_4 n$
- ...
- Outer loop terminates after last iteration ℓ.
- Iteration ℓ: 1 time inner loop with merge for $m = n$
  $c_2 + n/n(c_3 + nc_4) = c_2 + c_3 + c_4 n$

Bottom-Up Merge Sort: Analysis I

Assumption: $n = 2^k$ for some $k \in \mathbb{N}_{>0}$

Iterations of the outer loop (m for hi-lo+1):

- Iteration 1: $n/2$ times inner loop with merge for $m = 2$
  $c_2 + n/2(c_3 + 2c_4) = c_2 + 0.5c_3 n + c_4 n$
- Iteration 2: $n/4$ times inner loop with merge for $m = 4$
  $c_2 + n/4(c_3 + 4c_4) = c_2 + 0.25c_3 n + c_4 n$
- ...
- Outer loop terminates after last iteration ℓ.
- Iteration ℓ: 1 time inner loop with merge for $m = n$
  $c_2 + n/n(c_3 + nc_4) = c_2 + c_3 + c_4 n$

Total $T(n) \leq c_1 + \ell(c_2 + c_3 n + c_4 n) \leq \ell(c_1 + c_2 + c_3 + c_4)n$

Bottom-Up Merge Sort: Analysis II

What is the value of ℓ?

- In iteration i we have $m = 2^i$ for the merge step.
- In iteration ℓ we have $m = 2^\ell = n$ for the merge step.
- Since $n = 2^k$ we have $\ell = k = \log_2 n$.

With $c := c_1 + c_2 + c_3 + c_4$ we get $T(n) \leq cn \log_2 n$.

Bottom-Up Merge Sort: Analysis III

What if n is not a power of two, so $2^{k-1} < n < 2^k$?

- Nevertheless k iterations of the outer loop.
- Inner loop does not perform more operations.
- $T(n) \leq cnk = cn(\lfloor \log_2 n \rfloor + 1) \leq 2cn \log_2 n$ (for $k > 2$)

Bottom-Up Merge Sort: Analysis IV

Analogous argument possible for lower bound.
$\rightarrow$ Exercises

Theorem

Bottom-up merge sort has linearithmic running time, i.e. there are constants $c, c', n_0 > 0$, such that for all $n \geq n_0$:
$cn \log_2 n \leq T(n) \leq c' n \log_2 n$.

Linearithmic Running Time

Linearithmic running time $n \log_2 n$:
$\rightarrow$ twice as large input, slightly more than twice the running time

Linearithmic Running Time

Linearithmic running time $n \log_2 n$:
$\rightarrow$ twice as large input, slightly more than twice the running time

What does this mean in practice?

- Assumption: $c = 1$, one operation takes on average 10^{-8} sec.

Linearithmic Running Time

Linearithmic running time $n \log_2 n$:
$\rightarrow$ twice as large input, slightly more than twice the running time

What does this mean in practice?

- Assumption: $c = 1$, one operation takes on average 10^{-8} sec.
- With 1000 elements, we wait
  $10^{-8} \cdot 10^3 \log_2(10^3) \approx$ 0.0001 seconds.

Linearithmic Running Time

Linearithmic running time $n \log_2 n$:
$\rightarrow$ twice as large input, slightly more than twice the running time

What does this mean in practice?

- Assumption: $c = 1$, one operation takes on average 10^{-8} sec.
- With 1000 elements, we wait $10^{-8} \cdot 10^3 \log_2(10^3) \approx$ 0.0001 seconds.
- With 10 thousand elements $\approx$ 0.0013 seconds.

Linearithmic Running Time

Linearithmic running time $n \log_2 n$:
$\rightarrow$ twice as large input, slightly more than twice the running time

What does this mean in practice?

- Assumption: $c = 1$, one operation takes on average 10^{-8} sec.
- With 1000 elements, we wait
  $10^{-8} \cdot 10^3 \log_2(10^3) \approx$ 0.0001 seconds.
- With 10 thousand elements $\approx$ 0.0013 seconds.
- With 100 thousand elements $\approx$ 0.017 seconds.

Linearithmic Running Time

Linearithmic running time $n \log_2 n$:
$\rightarrow$ twice as large input, slightly more than twice the running time

What does this mean in practice?

- Assumption: $c = 1$, one operation takes on average 10^{-8} sec.
- With 1000 elements, we wait $10^{-8} \cdot 10^3 \log_2(10^3) \approx$ 0.0001 seconds.
- With 10 thousand elements $\approx$ 0.0013 seconds.
- With 100 thousand elements $\approx$ 0.017 seconds.
- With 1 million elements $\approx$ 0.2 seconds.

Linearithmic Running Time

Linearithmic running time $n \log_2 n$:
$\rightarrow$ twice as large input, slightly more than twice the running time

What does this mean in practice?

- Assumption: $c = 1$, one operation takes on average 10^{-8} sec.
- With 1000 elements, we wait $10^{-8} \cdot 10^3 \log_2(10^3) \approx$ 0.0001 seconds.
- With 10 thousand elements $\approx$ 0.0013 seconds.
- With 100 thousand elements $\approx$ 0.017 seconds.
- With 1 million elements $\approx$ 0.2 seconds.
- With 1 billion elements $\approx$ 299 seconds.

Linearithmic Running Time

Linearithmic running time $n \log_2 n$:
$\rightarrow$ twice as large input, slightly more than twice the running time

What does this mean in practice?

- Assumption: $c = 1$, one operation takes on average 10^{-8} sec.
- With 1000 elements, we wait $10^{-8} \cdot 10^3 \log_2(10^3) \approx$ 0.0001 seconds.
- With 10 thousand elements $\approx$ 0.0013 seconds.
- With 100 thousand elements $\approx$ 0.017 seconds.
- With 1 million elements $\approx$ 0.2 seconds.
- With 1 billion elements $\approx$ 299 seconds.

Running time $n \log_2 n$ not much worse than linear running time

Merge Sort with Cost Model I

Key comparisons

- Only in `merge`.
- Merging two ranges of length m and n requires in the best case $\min(n, m)$ and in the worst case $n + m - 1$ comparisons.
- With two ranges of roughly equal length, this is a linear number of comparisons, i.e., there are $c, c' > 0$ such that the number of comparisons is between cn and $c'n$.

$\rightarrow$ Number of key comparisons that is performed for sorting the entire input sequence is linearithmic in the length of the sequence (analogously to the runtime analysis).

Merge Sort with Cost Model II

Movements of elements

- Only in `merge`.
- $2n$ movements for sequence of length n.
- Total for merge sort linearithmic
  (analogously to key comparisons).

Summary

Summary

- Merge sort has linearithmic running time, key comparisons and movements of elements.