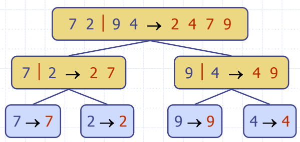
Merge Sort



Outline and Reading

- Divide-and-conquer paradigm (§4.1.1)
- Merge-sort (§4.1.1)
  - Algorithm
  - Merging two sorted sequences
  - Merge-sort tree
  - Execution example
  - Analysis
- Generic merging and set operations (§4.2.1)
- Summary of sorting algorithms (§4.2.1)

Divide-and-Conquer

- Divide-and conquer design paradigm:
  - Divide: divide the input data S in two (or more) disjoint subsets S_1 and S_2
  - Recur: solve the subproblems associated with S_1 and S_2
  - Conquer: combine the solutions for S_1 and S_2 into a solution for S
- Base case: directly solve and do not divide for "small" subproblem sizes (typically 0 or 1).
- Merge-sort is a sorting algorithm based on divide-and-conquer
- Like heap-sort
  - $O(n \log n)$ running time
- Unlike heap-sort
  - No auxiliary priority queue
  - Accesses data sequentially (suitable to sort data on a disk)

Merge-Sort

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

```
Algorithm mergeSort(S)
  Input sequence S with n
    elements
  Output sequence S sorted

  if n > 1
    (S1, S2) ← partition(S, n/2)
    S1 ← mergeSort(S1)
    S2 ← mergeSort(S2)
    S ← merge(S1, S2)
  return S
```


Partitioning a Sequence

- The divide step of merge-sort consists of partitioning input sequence S
- Use doubly linked list with head and tail pointer
- Then all sequence ADT operations take O(1) time.
- With n total elements, *partition* takes $O(n)$ time.

```
Algorithm partition(S, k)
  Input sequence S, with n items;
      k, partition size
  Output partition of S into S1 of size k
      and S2 of size n - k

  S1 ← empty sequence
  S2 ← empty sequence
  pos ← S.first()
  for i ← 1 to k do
    S1.insertLast(pos.element())
    pos ← S.after(pos)
  for i ← k + 1 to n do
    S2.insertLast(pos.element())
    pos ← S.after(pos)
  return (S1,S2)
```


Merging Two Sorted Sequences

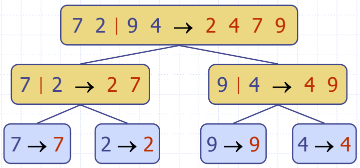
- The conquer step of merge-sort consists of merging two sorted sequences A and B
- Use doubly linked list with head and tail pointer
- Then all sequence ADT operations take O(1) time.
- With n total elements, *merge* takes $O(n)$ time.

```
Algorithm merge(A, B)
  Input sequence A and B, both sorted,
    with n total items combined
  Output sorted sequence of A ∪ B

  S ← empty sequence
  while ¬A.isEmpty() ∧ ¬B.isEmpty()
    if A.first().element() < B.first().element()
      S.insertLast(A.remove(A.first()))
    else
      S.insertLast(B.remove(B.first()))
  while ¬A.isEmpty()
    S.insertLast(A.remove(A.first()))
  while ¬B.isEmpty()
    S.insertLast(B.remove(B.first()))
  return S
```

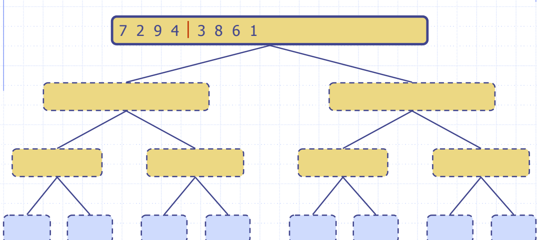
Merge-Sort Tree

- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences of size 0 or 1



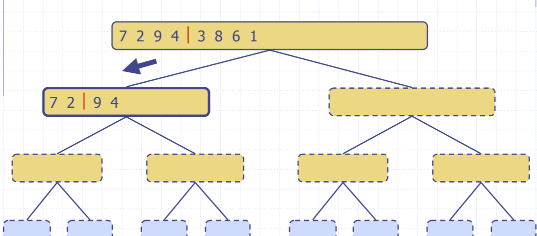
Execution Example

- Partition



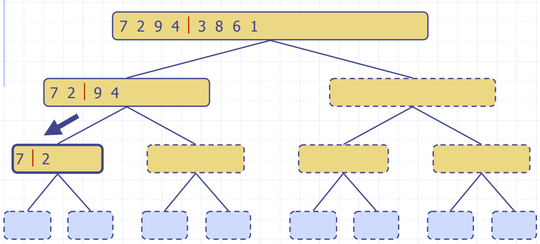
Execution Example (cont.)

- Recursive call, partition



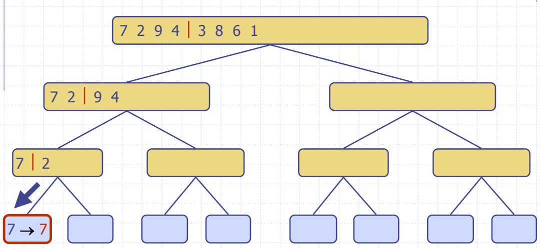
Execution Example (cont.)

- Recursive call, partition



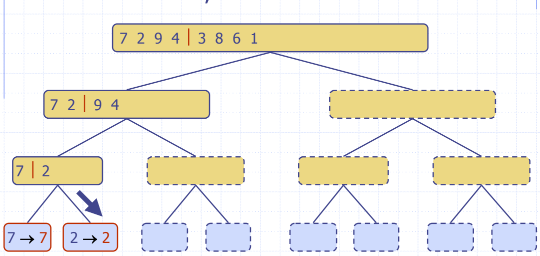
Execution Example (cont.)

- Recursive call, base case



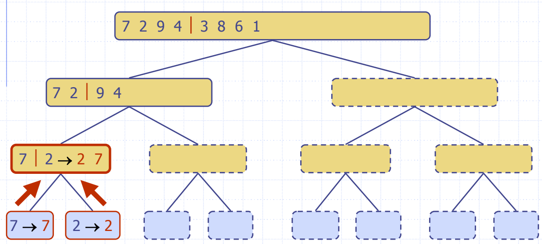
Execution Example (cont.)

- Recursive call, base case



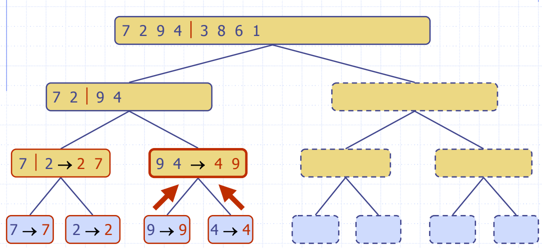
Execution Example (cont.)

- Merge



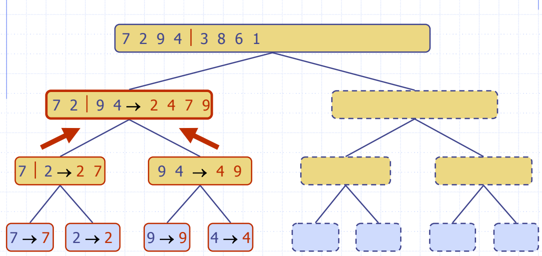
Execution Example (cont.)

- Recursive call, …, base case, merge



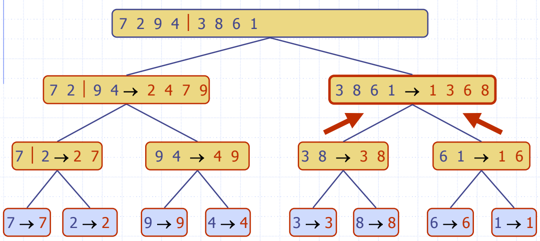
Execution Example (cont.)

- Merge



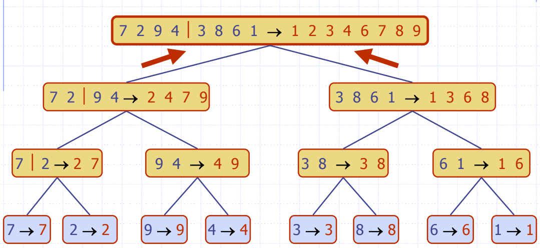
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

- Merge



Merge-Sort Analysis

- Use recurrence equation.

Algorithm *mergeSort*(S)
  Input sequence S with n elements
  Output sequence S sorted

  if $n > 1$
    $(S_1, S_2) \leftarrow partition(S, n/2)$
    $S_1 \leftarrow mergeSort(S_1)$
    $S_2 \leftarrow mergeSort(S_2)$
    $S \leftarrow merge(S_1, S_2)$
  return S

Merge-Sort Analysis

- Use recurrence equation.
- T(0) = T(1) = 2
- T(n) = cn + T(n/2) + T(n/2) + cn = 2cn + 2T(n/2)
- c is a constant.

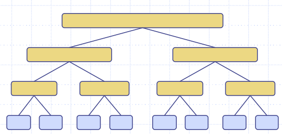
Algorithm *mergeSort*(S)
  Input sequence S with n elements
  Output sequence S sorted

  if $n > 1$
    $(S_1, S_2) \leftarrow partition(S, n/2)$
    $S_1 \leftarrow mergeSort(S_1)$
    $S_2 \leftarrow mergeSort(S_2)$
    $S \leftarrow merge(S_1, S_2)$
  return S

Analysis of Merge-Sort

- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is about $2cn \log n$, or $O(n \log n)$

depth	#calls	size	cost
0	1	n	$2cn$
1	2	$n/2$	$2cn$
i	2^i	$n/2^i$	$2cn$
…	…	…	

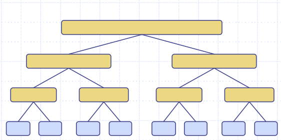


The Recursion Tree

- For solving divide-and-conquer recurrence relations:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

depth	T's	size	cost
0	1	n	bn
1	2	$n/2$	bn
i	2^i	$n/2^i$	bn
…	…	…	…



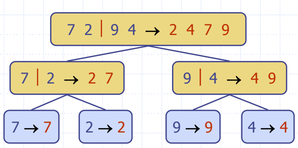
Total cost = $bn + bn \log n$

(last level plus all previous levels)

Summary of Sorting Algorithms

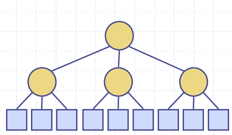
Algorithm	Time	Notes
selection-sort	$O(n^2)$	◆ slow ◆ in-place ◆ for small data sets (< 1K)
insertion-sort	$O(n^2)$	◆ slow ◆ in-place ◆ for small data sets (< 1K)
heap-sort	$O(n \log n)$	◆ fast ◆ in-place ◆ for large data sets (1K — 1M)
merge-sort	$O(n \log n)$	◆ fast ◆ sequential data access ◆ for huge data sets (> 1M)

Divide-and-Conquer



Divide-and-Conquer

- Analysis can be done using **recurrence equations**
- What would recurrence equation look like for this tree?



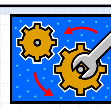
Recurrence Equation Analysis

- The conquer step of merge-sort consists of merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes at most bn steps, for some constant b.
- The basis case ($n < 2$) takes 2 steps.
- Therefore, if we let $T(n)$ denote the running time of merge-sort:

$$T(n) = \begin{cases} 2 & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

- We can therefore analyze the running time of merge-sort by finding a **closed form solution** to the above equation.
  - That is, a solution that has $T(n)$ only on the left-hand side.

Iterative Substitution

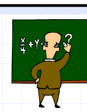


- In the iterative substitution, or "plug-and-chug," technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$\begin{aligned} T(n) &= 2T(n/2) + bn \\ &= 2(2T(n/2^2)) + b(n/2)) + bn \\ &= 2^2 T(n/2^2) + 2bn \\ &= 2^3 T(n/2^3) + 3bn \\ &= 2^4 T(n/2^4) + 4bn \\ &= \ldots \\ &= 2^i T(n/2^i) + ibn \end{aligned}$$

- Note that base, T(1)=2, case occurs when $n/2^i=1$. (Or i = log n).
- So, $T(n) = 2n + bn \log n$
- Thus, T(n) is O(n log n).

Solving recurrence equations



- Recurrence Trees (already shown)
- Iterative Substitution (already shown)
- Guess-and-Test Method (in book)
- Master Method (next)
  - does not apply to all recurrence equations!

Master Method

- Many divide-and-conquer recurrence equations have the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem: Note: ε, k are constants you pick.

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
   provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

Master Method, Example 1

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
     provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = 4T(n/2) + n$$

Solution: $\log_b a=2$, so case 1 says T(n) is $O(n^2)$.

Master Method, Example 2

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
     provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

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

Solution: $\log_b a=1$, so case 2 says T(n) is $O(n \log^2 n)$.

Master Method, Example 3

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
     provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = T(n/3) + n \log n$$

Solution: $\log_b a$=0, so case 3 says T(n) is O(n log n).

Master Method, Example 4

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
     provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = 8T(n/2) + n^2$$

Solution: $\log_b a$=3, so case 1 says T(n) is O(n^3).

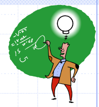
Master Method, Example 5

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
     provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = 9T(n/3) + n^3$$

Solution: $\log_b a$=2, so case 3 says T(n) is O(n^3).

Master Method, Example 6

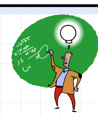


- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = T(n/2) + 1$$ (binary search)

Solution: $\log_b a=0$, so case 2 says T(n) is O(log n).

Master Method, Example 7

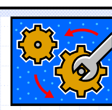


- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = 2T(n/2) + \log n$$ (heap construction)

Solution: $\log_b a=1$, so case 1 says T(n) is O(n).

Iterative "Proof" of the Master Theorem



- Using iterative substitution, let us see if we can find a pattern:

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ &= a(aT(n/b^2)) + f(n/b)) + bn \\ &= a^2T(n/b^2) + af(n/b) + f(n) \\ &= a^3T(n/b^3) + a^2f(n/b^2) + af(n/b) + f(n) \\ &= \ldots \\ &= a^{\log_b n}T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i) \\ &= n^{\log_b a}T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i) \end{aligned}$$

- We then distinguish the three cases as
  - The first term is dominant
  - Each part of the summation is equally dominant
  - The summation is a geometric series