Last Lecture

- Sorting
  - Selection sort
  - Insertion sort
  - Bubble sort
  - Improved bubble sort
- Time complexity of each sorting algorithm

A Problem

Given two sorted subsequences, how to merge them together to produce one sorted sequence?

CPT108 Data Structures and Algorithms

Lecture 10

Sorting

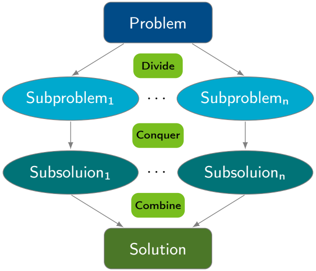
Merge Sort and Quick Sort

Outline

1. Sortings
2. Merge sort
3. Quick sort
   - Pick a Pivot
   - How to Partition

Merge sort and Quick sort

- Based on divide-and-conquer strategy:
  - Divide the problem into smaller, more manageable subproblems that looks similar to the initial problem
  - Then, solve these subproblem and put their solutions together to solve the original problem



Outline

Merge sort

Pseudocode

Given a sequence with N elements

- Divide the sequence into two smaller subsequences
- Sort each smaller subsequence *recursively*
- Merge the two sorted subsequences to produce one sorted sequence

Algorithm

$\textsc{Merge-Sort}(A, \mathit{left}, \mathit{right})$

if $\mathit{left} < \mathit{right}$

$\quad \mathit{mid} = (\mathit{left} + \mathit{right})/2$

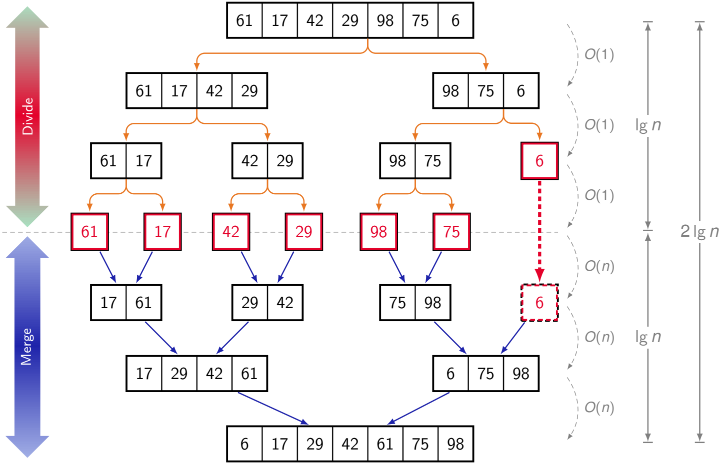
$\quad \textsc{Merge-Sort}(A, \mathit{left}, \mathit{mid})$

$\quad \textsc{Merge-Sort}(A, \mathit{mid} + 1, \mathit{right})$

$\quad \textsc{Merge}(A, \mathit{left}, \mathit{mid}, \mathit{right})$

Merge sort

An example



Merge sort

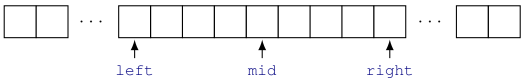
Exercise

21	21	48	16	51	60	73	24

- Questions:
  - How do we divide the sequence?
    How much time is needed?
  - How do we merge the two sorted sequences?
    How about time is needed

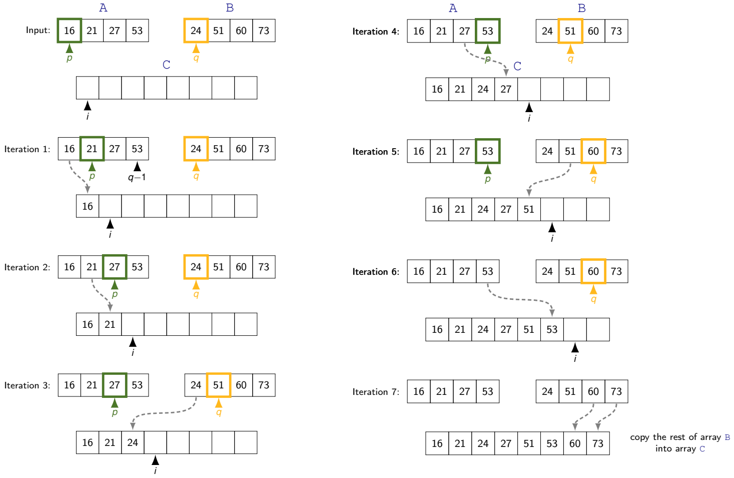
Merge sort: Divide

- If the sequence is given as an array `A[0,...,N−1]`
  - Dividing takes $O(1)$ time!
  - We can represent any subsequence of `A[0,...,N−1]` by two integers: `left` and `right` which index the two entries delimiting the subsequence
  - E.g., To divide `A[left,...,right]`, we compute `mid` $=$ (`left` $+$ `right`)/2 and obtain `A[0,...,mid]` and `A[mid+1,...,N−1]`



Merge sort: Merge

- How to merge?
  Input: Two sorted arrays A and B   Output: A sorted array C



Merge sort: Merge (cont.)

```
MERGE(A, p, q, r)
    // Input: Subarrays A[p, ..., l] and A[q, ..., r] s.t. p ≤ l = q − 1 < r
    // Output: A[p,...,r] is sorted
    // * T is a temporary array
    k = p; i = 0; l = q − 1
    while p ≤ l and q < r // copy the values from the two arrays to T
        if A[p] ≤ A[q]
            T[i] = A[p]; i = i + 1; p = p + 1
        else T[i] = A[q]; i = i + 1; q = q + 1
    while p ≤ l // copy the rest of the 1st array to T
        T[i] = A[p]; i = i + 1; p = p + 1
    while p ≤ l // copy the rest of the 2nd array to T
        T[i] = A[q]; i = i + 1; q = q + 1
    for i = k to r // copy back
        A[i] = T[i − k]
```

Clearly, `Merge` takes $O(m_1 + m_2)$ where m_1 and m_2 are the size of the two input arrays

Space requirement:

- Merging two sorted arrays requires linear extra memory
- Additional work to copy to the temporary array and back

Merge sort: Complexity

Let $T(N)$ be the worst-case running time of merge sort to sort N numbers.

For simplicity, we assume N is a power of 2, i.e., $N = 2^k$ or $k = \log N$, where k is a constant.

- Divide step: $O(1)$ time
- Conquer step: $2\ O(\frac{N}{2})$ time
- Combine step: $O(N)$ time

$\therefore$ Recurrence equation:

$$\begin{cases} T(1) = 1 \\ T(N) = 2T(\frac{N}{2}) + N \end{cases}$$

$$\begin{aligned}
\therefore T(N) &= 2T(\tfrac{N}{2}) + N \\
&= 2\left[2T(\tfrac{N}{4}) + \tfrac{N}{2}\right] + N \\
&= 4T(\tfrac{N}{4}) + 2N \\
&= 4\left[2T(\tfrac{N}{8}) + \tfrac{N}{4}\right] + 2N \\
&= 8T(\tfrac{N}{8}) + 3N \\
&= \cdots \\
&= 2^k T(\tfrac{N}{2^k}) + kN \\
&= N \cdot T(1) + kN \\
&= N + N\log N \\
&= O(N\log N)
\end{aligned}$$

Complexities

	Worst	Best
Selection sort	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n)$
Bubble sort	$O(n^2)$	$O(n^2)$
Improved bubble sort	$O(n^2)$	$O(n)$
Merge sort	$O(n \log n)$	$O(n \log n)$

Merge sort

Summary (Geeksforgeeks.org, 2024a)

Advantages

- Merge sort can easily *parallelized* to take advantage on multiple processors or threads
- Can be used in external sorting, where data to be sorted is too large to fit into memory
- *Guaranteed worst-case performance* ($O(n \log n)$) which means it performs well even on large datasets
  - Good to sort large datasets
  - Can be adapted to handle different input distributions, such as partially sorted

Drawbacks

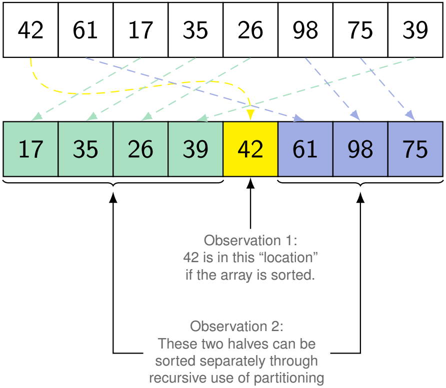
- *Space complexity*
  - It requires additional memory to store the sorted data during the sorting process
- *High overhead* for small datasets

Outline

1. Sortings
2. Merge sort
3. Quick sort
   - Pick a Pivot
   - How to Partition

Quick sort

Motivation



How could we use this operation for sorting?

Quick sort

- Another divide-and-conquer recursive algorithm, like merge sort
- ***Fastest*** known sorting algorithm in practice
- Average case: $O(N \log N)$
- Worst-case: $O(N^2)$
  - But the worst-case *rarely* happens

Quick sort: Algorithm

Pseudocode

Choose an element from the array as pivot
Reorder the array so that
  all elements with values less than the pivot come before it,
  while all elements with values greater than the pivot come after it
Recursively apply the above steps to the sub-array of elements
  with smaller and larger values

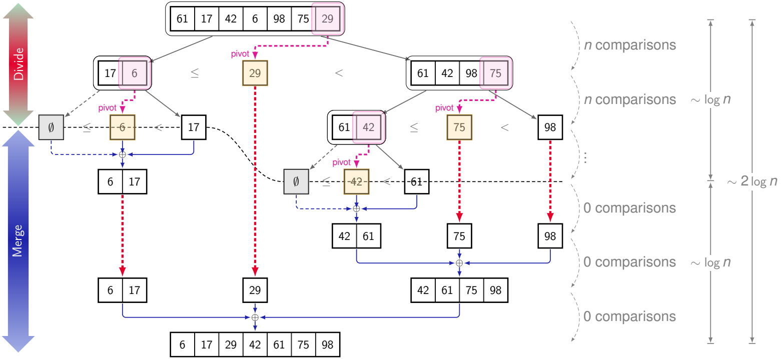
Algorithm

$\text{QUICKSORT}(A, p, r)$

if $p < r$)
    // Partition the subarray around the pivot, which ends up in A[q]
    $q = \text{PARTITION}(A, p, r)$
    $\text{QUICKSORT}(A, p, q - 1)$ // recursively sort the low side
    $\text{QUICKSORT}(A, q + 1, r)$ // recursively sort the high side

Quick sort

An example

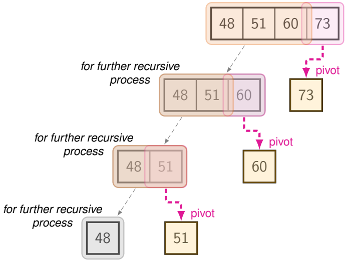


Quick sort

Exercise

21	21	48	16	51	60	73	24

Quick sort: Exercise



Observations:

- If the input items are already *sorted*, then only *one* item can be removed in each recursive call.
- Therefore, the number of recursive calls required, i.e., the depth of the tree, will become n, instead of $\log n$!

$\Rightarrow$ The computational complexity becomes $O(n \times n) = O(n^2)$! (analyse this case later)

Quick sort (cont.)

Two key steps

- How to pick a pivot?
- How to partition?

Quick sort

Pick a Pivot

- Picking a good pivot is necessary for the fast implementation of quick sort. However, it is often difficult to determine what a good pivot is.
- Common ways of choosing a pivot include:
  - Randomly select an element from the given array
  - Select the rightmost or leftmost element of the given array
  - Some implementations also used the median as the pivot element
- However, if the array is pre-ordered (or in reverse order)
  - All the elements go into one side of the array
  - Results in $O(N^2)$

Quick sort

How to Partition

Pseudocode

This is our goal!

Choose an element from the array as pivot
Reorder the array so that
    all elements with values less than the pivot come before it,
    while all elements with values greater than the pivot come after it
Recursively apply the above steps to the sub-array of elements
    with smaller and larger values

- If use additional array (not in-place) like merge sort
    - Straightforward to code
    - But inefficiency!

Quick sort

Partition (cont.)

IN-PLACE-PARTITION(A, p, r)

$x = A[r]$ // the pivot
$i = p - 1$ // highest index into the low side
for $j = p$ **to** $r - 1$ // process each element other than the pivot
    if $A[j] \leq x$ // does this element belong on the low side?
        $i = i + 1$ // index of a new slot in the low side
        exchange $A[i]$ with $A[j]$ // put this element there
exchange $A[i+1]$ with $A[r]$ // pivot goes just to the right of the low side
return $i + 1$ // new index of the pivot

	i	p, j							r
input:		2	8	7	1	3	5	6	4
		p, i	j						r
iteration 1:		2	8	7	1	3	5	6	4
		p, i		j					r
iteration 2:		2	8	7	1	3	5	6	4
		p, i			j				r
iteration 3:		2	8	7	1	3	5	6	4
		p	i			j			r
iteration 4:		2	1	7	8	3	5	6	4
		p		i			j		r
iteration 5:		2	1	3	8	7	5	6	4
		p		i				j	r
iteration 6:		2	1	3	8	7	5	6	4
		p		i					r
iteration 7:		2	1	3	8	7	5	6	4
		p		i					r
iteration 8:		2	1	3	4	7	5	6	8

Quick sort: Complexities

	Worst	Best
Selection sort	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n)$
Bubble sort	$O(n^2)$	$O(n^2)$
Improved bubble sort	$O(n^2)$	$O(n)$
Merge sort	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n^2)$	$O(n \log n)$

Quick sort: Complexity

Assumption:

- A random pivot
- No cutoff for small arrays

Running time:

- Pivot selection: constant time, i.e., $O(1)$
- Partitioning: linear time, i.e., $O(N)$
- Running time of the two recursive calls:

$$\begin{cases} T(1) = 1 \\ T(N) = T(i) + T(N-i-1) + cN \end{cases}$$

where c is a constant and i is the number of elements in the left partition

For simplicity, we assume N is a power of 2, i.e., $N = 2^k$ or $k = \log N$, where k is a constant.

Average/Best case

i.e., when the partition is balanced, we have $T(i) \approx T(N-i-1) = T(\frac{N}{2})$

$$\begin{aligned} \therefore\ T(N) &= 2T(\tfrac{N}{2}) + cN \\ &= 2\left[2T(\tfrac{N}{4}) + c\tfrac{N}{2}\right] + cN \\ &= 4T(\tfrac{N}{4}) + 2cN \\ &= \cdots \\ &= 2^k T(\tfrac{N}{2^k}) + kcN \\ &= N \cdot T(1) + kcN \\ &= N + c \cdot (N \log N) \\ &= O(N \log N) \end{aligned}$$

Quick sort: Complexity

Assumption:

- A random pivot
- No cutoff for small arrays

Running time:

- Pivot selection: constant time, i.e., $O(1)$
- Partitioning: linear time, i.e., $O(N)$
- Running time of the two recursive calls:

  $$\begin{cases} T(1) = 1 \\ T(N) = T(i) + T(N-i-1) + cN \end{cases}$$

  where c is a constant and i is the number of elements in the left partition

For simplicity, we assume N is a power of 2, i.e., $N = 2^k$ or $k = \log N$, where k is a constant.

Worst-case

i.e., when the partition is unbalanced

$$\begin{aligned} \therefore\ T(N) &= T(N-1) + cN \\ &= [T(N-2) + c(N-1)] + cN \\ &= T(N-2) + c\,[N + (N-1)] \\ &= \cdots \\ &= T(1) + c\Sigma_{i=2}^{N} i \\ &= T(1) + c\left[\frac{N(N+1)}{2} - 1\right] \\ &= O(N^2) \end{aligned}$$

Quick sort: Complexities

	Worst	Best
Selection sort	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n)$
Bubble sort	$O(n^2)$	$O(n^2)$
Improved bubble sort	$O(n^2)$	$O(n)$
Merge sort	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n^2)$	$O(n \log n)$

Quick sort

Summary (Geeksforgeeks.org, 2024b)

Advantages

- It is efficient on large datasets
- It has a low overhead, as it only requires a small amount of memory to function

Drawbacks

- It has a worst-case time complexity of $O(N^2)$, which occur when the pivot is chosen poorly
- It is not a stable algorithm, meaning that the relative order of elements will not be preserved in the sorted output

Quick sort

Some remarks

- For very small arrays, quick sort does not perform as well as insertion sort
  - how small depends on many factors, such as the time spent making a recursive call, the compiler, etc.
- Do *not* use quick sort recursively for small arrays
  - Instead, should use sorting algorithm that is efficient for small arrays, such as insertion sort

Reading

- Chapter 2 and 4, Cormen (2022)

References

Geeksforgeeks.org (2024a). *Merge Sort — Data Structure and Algorithm Tutorials*. Online: https://www.geeksforgeeks.org/merge-sort/. [last accessed: 20 Mar 2024].

— (2024b). *Quick Sort — Data Structure and Algorithm Tutorials*. Online: https://www.geeksforgeeks.org/quick-sort/. [last accessed: 20 Mar 2024].