COMP 3761: Algorithm Analysis and Design

Shidong Shan

BCIT

Overview

1. Divide-and-conquer design strategy
2. The Master Theorem
3. Applications of divide-and-conquer:
   - Mergesort
   - Quicksort
   - Binary search algorithm

Divide-and-conquer

- One of the most important and efficient algorithms in Computer Science
- General idea:
  1. Divide a problem's instance into two or more smaller instances
  2. Solve the smaller instances recursively (typically)
  3. Obtain a solution to the original (larger) instance by combining the solutions for the smaller instances.

Computing Sum

- Problem: Compute the sum of n numbers: $a_0, a_1, \ldots, a_{n-1}$.
- Solve it recursively by divide-and-conquer:
  1. If $n = 1$, return a_0; otherwise,
  2. divide the problem into **two** instances of the same problem.
  3. compute the sum of the first $\lfloor n/2 \rfloor$ numbers
  4. compute the sum of the remaining $\lceil n/2 \rceil$ numbers
  5. add the above two sums to get the sum in question

$$a_0 + \ldots + a_{n-1} = (a_0 + \ldots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \ldots + a_{n-1})$$

- Question: How efficient is this algorithm?

Divide-and-conquer recurrence

- given an instance of input size n
- divide the instance into b instances of size n/b, where the number of instances need to be solved is a
- a and b are constants, $a \geq 1$ and $b > 1$
- using smoothness rule, assume $n = b^k$, for some $k > 0$
- general divide-and-conquer recurrence for running time $T(n)$:

$$T(n) = aT(n/b) + f(n)$$

- $T(n)$: the running time for solving instance of size n
- $T(n/b)$: the running time for solving instance of size n/b
- $f(n)$: the time spent on dividing the problem into smaller ones and on combining their solutions, i.e., the recurrence overhead.

Master Theorem

If $f(n) \in \Theta(n^d)$, $d \geq 0$,

$$T(n) = aT(n/b) + f(n)$$

then the order of growth for $T(n)$ is:

- $T(n) \in \Theta(n^d)$, if $a < b^d$
- $T(n) \in \Theta(n^d \log n)$, if $a = b^d$
- $T(n) \in \Theta(n^{\log_b a})$, if $a > b^d$

Note: The analogous results hold for the O and Ω notations.

Master Theorem Examples

Using the Master Theorem to find the order of growth for the solutions of the following recurrences:

a. $T(n) = 4T(n/2) + n$, $T(1) = 1$
b. $T(n) = 4T(n/2) + n^2$, $T(1) = 1$
c. $T(n) = 4T(n/2) + n^3$, $T(1) = 1$

Mergesort

1. split array $A[0..n-1]$ in two about equal halves
2. make copies of each half in arrays B and C
3. sort arrays B and C recursively
4. merge sorted arrays B and C into array A to produce a sorted array.

Pseudocode of Mergesort

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    $Mergesort(B[0..\lfloor n/2 \rfloor - 1])$
    $Mergesort(C[0..\lceil n/2 \rceil - 1])$
    $Merge(B, C, A)$

Merging two sorted arrays

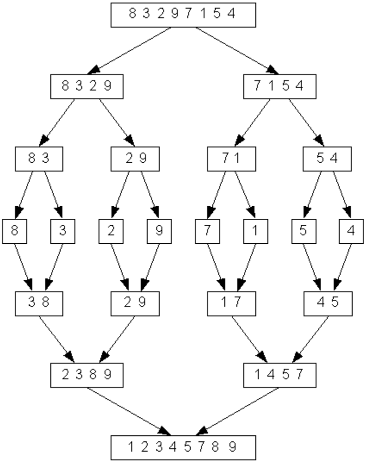
1. Starting with the first elements in the arrays being merged
2. Compare the first elements in the remaining unprocessed portions of the arrays
3. Copy the smaller of the two elements into the array under construction, and increment the index to its immediate next element
4. Repeat the process until one array is exhausted
5. Copy the remaining unprocessed elements from the other array to the end of the new array.

Pseudocode of Merge

ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
  if $B[i] \leq C[j]$
    $A[k] \leftarrow B[i];\ i \leftarrow i+1$
  else $A[k] \leftarrow C[j];\ j \leftarrow j+1$
  $k \leftarrow k+1$
if $i = p$
  copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Example: apply Mergesort to sort list: $8, 3, 2, 9, 7, 1, 5, 4$



Analysis

- Basic operation: key comparisons
- Recurrence relation:
  $$C(n) = 2C(n/2) + C_{merge}(n), \text{ for } n > 1,\ c(1) = 0$$
- $C_{merge}(n)$: number of key comparisons required during merging: $C_{merge}(n) = n - 1$ in the worst case.
- $C_{worst}(n) = 2C_{worst}(n/2) + n - 1$, for $n > 1$, $c(1) = 0$
- Apply Master Theorem, $C_{worst}(n) \in \Theta(n \log n)$

Comments on Mergesort

- Time efficiency: $\Theta(n \log n)$
- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting: $\approx \lceil n \log_2 n - 1.44n \rceil$
- Mergesort is stable
- Extra space requirement: $\Theta(n)$ (not in-place)
- In place merging is possible but quite complicated to implement
- Can be implemented without recursion (bottom-up approach).

Quicksort

- In practice, one of the fastest sorting algorithms
- Comparison-based sorting algorithm
- Based on divide-and-conquer approach
- Different from Mergesort:
  - Quicksort divides elements according to their value
  - Mergesort divides elements according to their position

Quicksort algorithm

- Select a pivot (partitioning element)
- Partitioning process: rearrange the list so that
  - s is the split position of the list
  - all the elements in the first s positions $\leq$ the pivot
  - all the elements in the remaining $n - s$ positions $\geq$ the pivot
- After a partition process, the pivot is in its final position in the sorted array
- Recursively sort the two subarrays by partitioning

Pseudocode

Algorithm Quicksort($A[l..r]$)

if $l < r$

$s \leftarrow Partition(A[l..r])$ // s is a split position

Quicksort($A[l..s-1]$)

Quicksort($A[s+1..r]$)

Pseudocode of partition using first element as a pivot

Algorithm $Partition(A[l..r])$

//Partitions a subarray by using its first element as a pivot

//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right

// indices l and r ($l < r$)

//Output: A partition of $A[l..r]$, with the split position returned as

// this function's value

$p \leftarrow A[l]$

$i \leftarrow l;\quad j \leftarrow r+1$

repeat

    repeat $i \leftarrow i+1$ **until** $A[i] \geq p$

    repeat $j \leftarrow j-1$ **until** $A[j] \cdot p$

    swap$(A[i], A[j])$

until $i \geq j$

swap$(A[i], A[j])$ //undo last swap when $i \geq j$

swap$(A[l], A[j])$

return j

Operation of Quicksort

Sort the following list by Quicksort (partition by using the first element as the pivot)

5 3 1 9 8 2 4 7

Quicksort: time complexity

- Basic operation: comparison
- Compare each element with the pivot in partition
- Best case: split in the middle, $\Theta(n \log n)$

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for} \quad n > 1,$$

- Worst case: given a sorted array, $\Theta(n^2)$
  partition always results in one subarray with only one element
  1st step: n comparisons
  2nd step: $n - 1$ comparisons
  3rd step: $n - 2$ comparisons
  ...

$$C_{worst}(n) = n + (n-1) + ... + 2 + 1 \in \Theta(n^2).$$

- Average case: random arrays $\Theta(n \log n)$

Quicksort: comments

- Quicksort is not stable
- In the average case, Quicksort usually outperforms Mergesort.
- Quicksort improvements:
  - better pivot selection: median of three partitioning
  - switch to a simpler sort on small subarrays
  - elimination of recursion: nonrecursive quicksort

Recursive binary search

- Problem: searching a key in a sorted array
- Input: $A[0..n-1], K$
- Output: index of K in $A[0..n-1]$, or -1 if K is not found.
- **Algorithm** $BSRec(A[l..r], K)$

  if $l > r$ return -1

  else $m \leftarrow \lfloor (l+r)/2 \rfloor$

    if $K = A[m]$,return m;

    else if $K < A[m]$, return $BSRec(A[l..m-1], K)$

    else return $BSRec(A[m+1, r], K)$.

Binary search: nonrecursive

Algorithm Nonrecursive BinarySearch($A[0..n-1], K$)

$l \leftarrow 0; \quad r \leftarrow n-1$

while $l \leq r$ do

$m \leftarrow \lfloor (l+r)/2 \rfloor$

if $K = A[m]$, return m

else if $K < A[m]$, $r \leftarrow m-1$

else $l \leftarrow m+1$

return -1

Analysis

- Worst-case recurrence:

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1, \quad \text{for } n > 1, \quad C(1) = 1$$

- Time efficiency: $C_{worst}(n) = \lceil \log_2(n+1) \rceil$
- This is VERY fast: e.g., $C_{worst}(10^6) = 20$
- Optimal for searching a sorted array
- Limitations: must be a sorted array (not linked list)
- A degenerate case of divide-and-conquer
- In fact, a decrease-by-half algorithm.

Exercises

- 4.1: #2, 3
- 4.2: #5, 8
- 4.3: #8, 9
- Read Section 1.4: Fundamental Data Structures on Graphs and Trees before the next class!