Analysis of Algorithms
- Recurrences (cont.) -

Andreas Ermedahl
MRTC (Mälardalens Real Time Research Center)
andreas.ermedahl@mdh.se
Autumn 2004

Recurrences: Rehearsal

- Recurrences appear frequently in running time formulas for recursive algorithms
- Three methods presented for solving such recurrences:
  - The Substitution method - guess a solution and use mathematical induction to show that it works
  - The Recursion-tree method - convert the recurrence into a tree whose nodes represent costs incurred at various levels of the recursion
  - The Master method - provides bounds for recurrences of the form: $T(n) = aT(n/b) + f(n)$ where constants $a \geq 1$, $b > 1$ and $f(n)$ is a given function
- We will investigate the Master Method in more detail

The Master method

- "Cookbook" method to solve recurrences on form: $T(n) = aT(n/b) + f(n)$
  - Suitable for $n > 0, a \geq 1$ and $b > 1$
  - For solving the kind of recurrences which often appear for divide and conquer algorithms
    - $a$ is the number of subproblems created
    - $n/b$ is the size of each subproblem
    - $f(n)$ is a function for the cost for dividing problem into subproblems and combining their solutions
- Can be applied in three different cases
  - Depends on how $n^{\log_b a}$ compares with $f(n)$
    - Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$
      - Gives solution: $T(n) = \Theta(n^{\log_b a})$
    - Case 2: $f(n) = \Theta(n^{\log_b a})$
      - Gives solution: $T(n) = \Theta(n^{\log_b a} \log n)$
    - Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$
      - and $f(n)$ satisfies that $af(n/b) \leq cf(n)$ for some $c < 1$
      - Gives solution: $T(n) = \Theta(f(n))$

The Master method

- We use a recurrence tree to understand Master Method
  - Total cost for tree is sum of cost for all nodes
  - Depending on what cost that dominates the total cost one of the three cases can be applied
  - Case 1: Cost for leaves dominate
  - Case 2: Both costs equally important
  - Case 3: Cost for root dominates
- Cost depends on value of $a$, $b$ and $f(n)$
- Cost for leaves is $O(n^{\log_b a})$
- Cost for internal nodes
  - Case 2 applies since $\log_b n^{\log_b a} = \log_b n^{\log_b a}$
- Cost for leaves is $O(n^{\log_b a})$

The Master method

- The Master Method gives tight bounds for $T(n) = aT(n/b) + f(n)$ in three cases:
  - Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$
    - Gives solution: $T(n) = \Theta(n^{\log_b a})$
  - Case 2: $f(n) = \Theta(n^{\log_b a})$
    - Gives solution: $T(n) = \Theta(n^{\log_b a} \log n)$
  - Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$
    - and $f(n)$ satisfies that $af(n/b) \leq cf(n)$ for some $c < 1$
    - Gives solution: $T(n) = \Theta(f(n))$
- Cases depend on how $f(n)$ compares with $n^{\log_b a}$
  - Roughly speaking:
    - Case 1 holds when: $f(n) < n^{\log_b a}$
    - Case 2 holds when: $f(n) = n^{\log_b a}$
    - Case 3 holds when: $f(n) > n^{\log_b a}$
  - But only roughly speaking!
  - The cases do not cover all possible $f(n)$ values
    - Case 1: $f(n)$ must be polynomially smaller than $n^{\log_b a}$
    - Case 3: $f(n)$ must be polynomially larger than $n^{\log_b a}$ and fulfill extra "regularity" condition
    - There are "gaps" between case 1 and 2 and between case 2 and 3 not covered by theorem
- Means that $f(n)$ must be smaller than $n^{\log_b a}$ by a factor of $n^k$ for constant $k > 0$
Master method: examples

- Examples of recurrences solvable by the Master method:
  - $T(n) = T(n/2) + 1$
  - $T(n) = 2T(n/2) + n$
  - $T(n) = 2T(n/2) + n^2$
- Example of recurrence not solvable by the Master method:
  - $T(n) = T(n/2) + \log n$

Master Method: Example

- The recurrence for MERGESORT is:
  $T(n) = 2T(n/2) + \Theta(n)$
- We set values as: $a=2$, $b=2$ and $f(n)=\Theta(n)$
  - We have that $n^{\log_2 2} = n^{1/2} = n$
  - Thus, $f(n)=\Theta(n^{\log_2 b})$ and case 2 applies
- We directly obtain $T(n) = \Theta(n^{\log_2 a} \log n)$
  - $= \Theta(n \log n)$
- Much easier!!! ☺

More Examples

- $T(n) = 5T(n/2) + \Theta(n^2)$
  - $\log_2 5 \cdot 2 = 2$ for some constant $\epsilon > 0$
  - We can use Case 1 $\Rightarrow T(n) = \Theta(n^{\log_2 5})$
- $T(n) = 5T(n/2) + \Theta(n^3)$
  - $\log_2 5 + \epsilon = 3$ for some constant $\epsilon > 0$
  - Check regularity condition:
    - $a f(n/b) = 5(n/2)^3 = 5n^3/8 \leq cn^3$ for $c = 5/8 < 1$
  - We can use Case 3 $\Rightarrow T(n) = \Theta(n^3)$

Master Method

- The proof for the Master Method is given in section 4.4
  - Uses recursion tree and induction techniques
- You do not need to understand the proof to apply the method
  - Just skim through section 4.4

Analysis of Algorithms

- Heapsort -

Andreas Ermedahl
MRTC (Mälardalens Real Time Research Center)
andreas.ermedahl@mdh.se
Autumn 2004

Sorting: Motivation

Why do comp. scientists never stop talking about sorting?
- Computers spend most time on organizing and shuffle around data
  - Small fraction spent on arithmetic computations
  - Sorting is the major part of this data maintenance
- Sorting is the best studied problem in computer science, with a variety of different algorithms known
- Most of the interesting ideas we will encounter in the course can be taught in the context of sorting
  - Examples: divide-and-conquer, randomized algorithms, and upper bounds.
Examples of sorting algorithms

- **Insertion Sort**: WC: O(n²) – BC: O(n) – in-place
- **Bubble Sort**: O(n²) – in-place – comparison
- **Merge Sort**: O(n log n) – not in-place - comparison
- **Heap Sort**: O(n log n) – in-place - comparison
- **Quick Sort**: WC: O(n²) – Avg: O(n log n) – in-place - comparison
- **Count Sort**: O(n) – in-place – no comparison
- **Bucket Sort**: O(n) – in-place – no comparison

In-place: only a constant number of input elements are stored outside array

Comparison sort: determines sorted order by comparing elements

Comparaison sorts have lower bound of Ω(n log n)

Some algorithms have linear exec time under certain conditions

In-place: only a constant number of input elements are stored outside array

Comparasion sort: determines sorted order by comparing elements

Comparaison sorts have lower bound of Ω(n log n)

Some algorithms have linear exec time under certain conditions

**HEAPSORT**

- O(n log n) worst case running time.
- Sorts in place: only O(1) extra memory needed
- Thus, HEAPSORT combines the best of INSERTIONSORT and MERGESORT

Two alternate ways to illustrate a heap

The heap data structure

- **Heapsort** uses a smart data structure, the (binary) heap
  - Useful for sorting, but also for other purposes like priority queues
  - Has nothing to do with garbage-collected memory areas
- A heap is a kind of **binary tree**
  - Balanced tree, where all leaves are "shifted leftmost" in the tree
  - Represented by an array A[1..n]

**The binary heap data structure**

- An array A storing a heap has two attributes:
  - length[A], size of A
  - heap_size[A], the size of the tree stored in A (which may be less than size)
- Each element (node) has two attributes:
  - position in the array, and
  - a value

**The Heap Property**

- For every node i (other than the root): A[Parent[i]] ≥ A[i]
- In other words:
  - The biggest element is stored at the root, A[1]
  - A parent never has a smaller value than its two children
- Finding parents and children is easy:
  - Parent(i) = ⌊i/2⌋
  - Left(i) = 2i
  - Right(i) = 2i + 1
- Above properties are for a MAX HEAP
  - A MIN-HEAP stores smallest value in the root

For every node i except root: A[Parent[i]] ≤ A[i]
Basic operations supported

- **Heapify** – “maintain the heap property”, i.e., restore a “disturbed” heap.
- **Build Heap** – build a heap from scratch with the heap maintenance routine.
- **Heap Sort** – sort a sequence using the maintenance routine (on a heap built from scratch).

The MAX-HEAPIFY procedure

- Routine to restore a (sub)heap where its subtrees are heaps
- Handles the case that the root violates the heap property (i.e., holds a smaller value than its children)
- Idea: swap root with child holding largest value
  - Must be done recursively on subtree whose root was swapped
    - since subtree now may violate the heap property

Analysis of MAX-HEAPIFY

- Best-case occurs when no swap is performed
  - Best-case time: $T_B(n) = O(1)$
  - $T(n) = O(1)$
- Worst-case occurs when we swap all elements in branch starting at root $A[1]$ and ending in leaf at max tree level
  - A full binary tree with $n$ nodes has $\log_2 n + 1$ levels and $\log_2 n$ height
  - At most 3 levels $= 1 + \log_2 n$ swaps can be performed
  - Worst-case time: $T_W(n) = O(\log n)$
  - $T(n) = O(\log n)$

Constructing a heap

- We use MAX-HEAPIFY to build a heap from an unstructured array
  - Elements in the resulting array should all fulfill heap property

Idea for algorithm:

- Create the heap bottom-up:
  - Start with leaves, then their parents, ....
  - The leaves directly have the heap property
- When adding a parent the resulting subtree might violate the heap property
  - Run MAX-HEAPIFY on the created subtree
Example

**BUILD-MAX-HEAP(A, n)**
for i ← ⌈n/2⌉ downto 1
    do MAX-HEAPIFY(A, i, n)

A[3] doesn’t fulfil heap property
A[2]=1 was moved down to A[10]
A[1]=4 was moved down to A[9]
Resulting tree fulfills heap property

Input array A

Analysing BUILD-MAX-HEAP

- Worst case time of BUILD-MAX-HEAP?
  - Simple upper bound: \( O(n \lg n) \)
  - Since \( n/2 \) calls to MAX-HEAPIFY, each \( O(\lg n) \)

- But this is not tight!
  - Many calls to MAX-HEAPIFY are on subtrees whose size \( \leq n \)
  - Time to for a run of MAX-HEAPIFY is linear in the height of the node it’s run upon
  - We should be able to obtain a tighter bound!

Height of Nodes

- Height of a node = longest distance from a leaf
- For a complete binary tree, there are \( \lceil n/2^{h+1} \rceil \) nodes of height \( h \)
- Example: \( n = 15 \) gives tree:
  - \( \lceil 15/2 \rceil = 8 \) leaves
  - \( \lceil 15/4 \rceil = 4 \) parents to leaves
  - \( \ldots \)
  - \( \lceil 15/16 \rceil = 1 \) root node

The HEAPSORT algorithm

1. BUILD-MAX-HEAP(A)
2. for i ← heap[1] downto 2
   3. exchange heap[i] ← heap[i-1]
   4. heapsize(A) ← heapsize(A) - 1
5. MAX-HEAPIFY(A, 1)

- First, create a heap incorporating all elements in \( A[1..n] \)
- After line 1, the maximum element is stored in \( A[1] \)
- Exchange it with \( A[n] \), then “disconnect” \( A[n] \) from heap
- Restore heap-property in \( A[1..n-1] \) with MAX-HEAPIFY
- Now the second largest element is in \( A[1] \) again, etc.

Example HEAPSORT

a) After BUILD-MAX-HEAP:

New heap constructed:

New heap constructed:

New heap constructed:

e) Only one element left
Stop:
HEAP-SORT

After running BUILD-MAX-HEAP
A[1] = 16

After changing A[1] and a[9], removing a[9] = 14 from heap, and rebuilding heap for remaining nodes
A[1] = 10

Analyzing HEAPSORT

- Call to BUILD-MAX-HEAP takes time O(n)
- n-1 calls to MAX-HEAPIFY
- Each call MAX-HEAPIFY takes O(lg n)
- Result: O(n) + (n-1) * O(lg n) = O(n lg n)

Priority Queues

- Priority queue: a data structure (representing a set S) in which each element x has a priority (or key k)
- Can be used in operating systems (job scheduling) or in communication (buffers with priorities on messages)
  - Min priority queue useful for event driven scheduling
- Three basic operations supported:
  - insert(S, x): inserts element x into set S
  - maximum(S): returns element of S with largest key
  - extract-max(S): removes and returns element with largest key
- Heaps can be used to represent priority queues!

Priority Queues: Implementation

- insert(A, x) = HEAP-INSERT(A, key(x)) where
  - HEAP-INSERT(A, key)
    1. heap-size[A] = heap-size[A] + 1
    2. i = heap-size[A]
    3. while i > 1 and A[parent(i)] < key do
      5. i = parent(i)
    6. A[i] = key

Summary

- HEAPSORT:
  - O(n lg n) worst-case running time.
  - Sorts in place: only Θ(1) extra memory needed
  - Based on comparisons between elements
  - Can be used to implement efficient priority queues
- Next lecture: QUICKSORT
  - A well-implemented QUICKSORT usually beats HEAPSORT in practice
The End!