CSE 373 Analysis of Algorithms
Fall 2016
Instructor: Prof. Sael Lee

LEC07 SORTING III: MERGESORT – SORTING DIVIDE-AND-CONQUER

Lecture slide courtesy of Prof. Steven Skiena

PROBLEM OF THE DAY

Problem: Given an array-based heap on n elements and a real number x, efficiently determine whether the kth smallest element in the heap is greater than or equal to x.

Your algorithm should be $O(k)$ in the worst-case, independent of the size of the heap.

Hint: you do not have to find the kth smallest element; you need only determine its relationship to x.

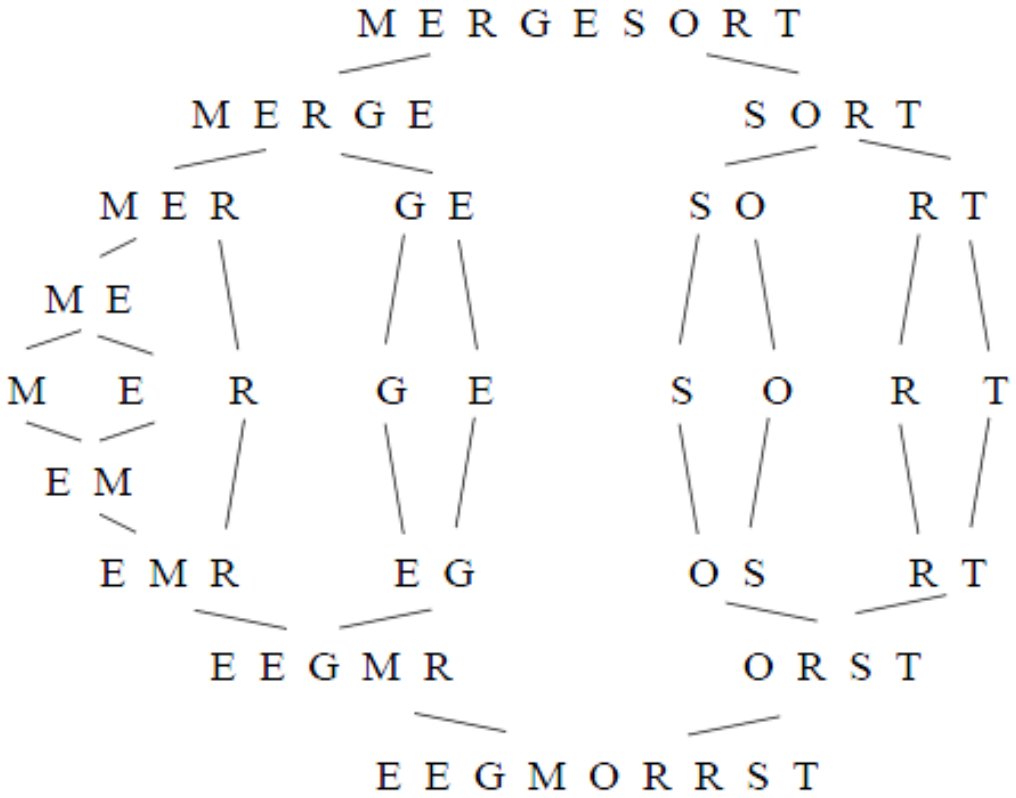
MERGESORT

- Recursive algorithms are based on reducing large problems into small ones.
- A nice recursive approach to sorting involves partitioning the elements into sub-groups, sorting each of the smaller problems recursively, and then interleaving the two sorted lists to totally order the elements.

MERGESORT IMPLEMENTATION

```
mergesort(item type s[], int low, int high)
{
        int i;                  /* counter */
        int middle;             /* index of middle element */
        if (low < high) {
                middle = (low+high)/2;
                mergesort(s,low,middle);
                mergesort(s,middle+1,high);
                merge(s, low, middle, high);
        }
}
```

MERGESORT ANIMATION



MERGING SORTED LISTS

- The efficiency of Mergesort depends upon how efficiently we combine the two sorted halves into a single sorted list.
- This smallest element can be removed, leaving two sorted lists behind, one slightly shorter than before.
- Repeating this operation until both lists are empty merges two sorted lists (with a total of n elements between them) into one, using at most n - 1 comparisons or O(n) total work
- Example: A = {5; 7; 12; 19} and B = {4; 6; 13; 15}

BUFFERING

- Although mergesort is *O(nlgn),* it is inconvenient to implement with arrays, since we need extra space to merge the lists.
  - Merging (4; 5; 6) and (1; 2; 3) would overwrite the first three elements if they were packed in an array.
- Writing the merged list to a buffer and recopying it uses extra space but not extra time (in the big Oh sense).

Challenging turns out to be the details of how the merging is done. We must put our merged array somewhere to avoid losing an element by overwriting it in the course of the merge.

we first copy each subarray to a separate queue

Then merge these elements back into the array.

```
merge(item_type s[], int low, int middle, int high)
{
  int i;                    /* counter */
  queue buffer1, buffer2; /* buffers to hold elements for merging */

  init_queue(&buffer1);
  init_queue(&buffer2);

  for (i=low; i<=middle; i++) enqueue(&buffer1,s[i]);
  for (i=middle+1; i<=high; i++) enqueue(&buffer2,s[i]);

  i = low;
  while (!(empty_queue(&buffer1) || empty_queue(&buffer2))) {
        if (headq(&buffer1) <= headq(&buffer2))
                s[i++] = dequeue(&buffer1);
        else
                s[i++] = dequeue(&buffer2);
  }

  while (!empty_queue(&buffer1)) s[i++] = dequeue(&buffer1);
  while (!empty_queue(&buffer2)) s[i++] = dequeue(&buffer2);
}
```

EXTERNAL SORTING

- Which O(nlogn) algorithm you use for sorting doesn't matter much until n is so big the data does not fit in memory.
- Mergesort proves to be the basis for the most efficient *external sorting* programs.(no random access needed)
- Disks are much slower than main memory, and benefit from algorithms that read and write data in long streams – not random access.

BREAKING PROBLEMS DOWN INTO SMALLER PROBLEMS

- *Dynamic Programming (ch8)* typically removes one element from the problem, solves the smaller problem, and then uses the solution to this smaller problem to add back the element in the proper way.
- *Divide-and-conquer* splits the problem in (say) halves, solves each half, then stitches the pieces back together to form a full solution.

DIVIDE AND CONQUER

- Divide and conquer is an important algorithm design technique using in mergesort, binary search the fast Fourier transform (FFT), and Strassen's matrix multiplication algorithm.
- We divide the problem into two smaller subproblems, solve each recursively, and then meld the two partial solutions into one solution for the full problem.
- When merging takes less time than solving the two subproblems, we get an efficient algorithm.

MATRIX MULTIPLICATION (BRUTE FORCE)

2.5.4 Matrix Multiplication

```
for (i=1; i<=x; i++)
        for (j=1; j<=y; j++) {
                C[i][j] = 0;
                for (k=1; k<=z; k++)
                        C[i][j] += A[i][k] * B[k][j];
        }
```

$$M(x,y,z) = \sum_{i=1}^{x}\sum_{j=1}^{y}\sum_{k=1}^{z} 1$$

$$M(x,y,z) = \sum_{i=1}^{x}\sum_{j=1}^{y} z$$

$$M(x,y,z) = \sum_{i=1}^{x} yz$$

algorithm is $O(xyz)$

multiplication: $\Theta(n^3)$

addition: $\Theta(n^3)$

MATRIX MULTIPLICATION (DIVIDE-CONQUER RECURSIVE ALGORITHM)

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

$$= \begin{bmatrix} A_{00} * B_{00} + A_{01} * B_{10} & A_{00} * B_{01} + A_{01} * B_{11} \\ A_{10} * B_{00} + A_{11} * B_{10} & A_{10} * B_{01} + A_{11} * B_{11} \end{bmatrix}$$

A, B: n by n matrices;

A_{ij}, B_{ij}: n/2 by n/2 matrices, where $i, j \in \{0, 1\}$

recurrence relations:

multiplication: $M(n) = \Theta(n^3)$

addition: $A(n) = \Theta(n^3)$

STRASSEN'S MATRIX MULTIPLICATION

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

$$= \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}$$

$M_1=(A_{00}+A_{11})*(B_{00}+B_{11})$
$M_2=(A_{10}+A_{11})*B_{00}$
$M_3=A_{00}*(B_{01}-B_{11})$
$M_4=A_{11}*(B_{10}-B_{00})$
$M_5=(A_{00}+A_{01})*B_{11}$
$M_6=(A_{10}-A_{00})*(B_{00}+B_{01})$
$M_7=(A_{01}-A_{11})*(B_{10}+B_{11})$

For matrices of size $N = 2^n$, let $f(n)$ be the number of operations for a $2^n \times 2^n$ matrix. Recursive formula for Strassen algorithm:

$T(n) = 7T(n-1) + O(4^n)$

$$T(1) = 1 \qquad (\text{assume } N = 2^k)$$
$$T(N) = 7T(N/2)$$
$$T(N) = 7^k T(N/2^k) = 7^k$$
$$T(N) = 7^{\log N} = N^{\log 7} = N^{2.81}$$

RECURRENCE RELATIONS

- Many divide-and-conquer algorithms have time complexities that are naturally modeled by recurrence relations.
- *Recurrence Relation* is an equation that is defined in terms of itself.
  - Example: The Fibonacci numbers are described by the recurrence relation $F_n = F_{n-1}+F_{n-2}$
  - Any polynomial can be represented by a recurrence

    linear function: $a_n = a_{n-1} + 1$, $a_1 = 1 \rightarrow a_n = n$

    exponential: $a_n = 2a_{n-1}$, $a_1 = 1 \rightarrow a_n = 2^{n-1}$

DIVIDE-AND-CONQUER RECURRENCES

Divide-and-conquer algorithms tend to break a given problem into some number of smaller pieces (say a), each of which is of size n/b. Further, they spend $f(n)$ time to combine these subproblem solutions into a complete result.

Let $T(n)$ denote the worst-case time the algorithm takes to solve a problem of size n. Then $T(n)$ is given by the following recurrence relation:

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

RECURRENCE : *MERGESORT*

The running time behavior of mergesort is governed by the recurrence

$$T(n) = 2T(n/2) + O(n),$$

- since the algorithm divides the data into equal-sized halves and
- then spends linear time merging the halves after they are sorted.

In fact, this recurrence evaluates to $T(n) = O(n \lg n)$, just as we got by our previous analysis.

RECURRENCE: *BINARY SEARCH*

The running time behavior of binary search is governed b y the recurrence

$$T(n) = T(n/2) + O(1),$$

since at each step we spend constant time to reduce the problem to an instance half its size.

In fact, this recurrence evaluates to $T(n) = O(\lg n)$, just as we got by our previous analysis.

FAST HEAP CONSTRUCTION

- *Fast Heap Construction* reduces to the recurrence relation

$$T(n) = 2T(n/2) + O(\lg n).$$

- Since the bubble down method of heap construction built an n-element heap by constructing two $n/2$ element heaps and then merging them with the root in logarithmic time.

- In fact, this recurrence evaluates to $T(n) = O(n)$, just as we got by our previous analysis.

MASTER THEOREM

Divide-and-conquer recurrences of the form

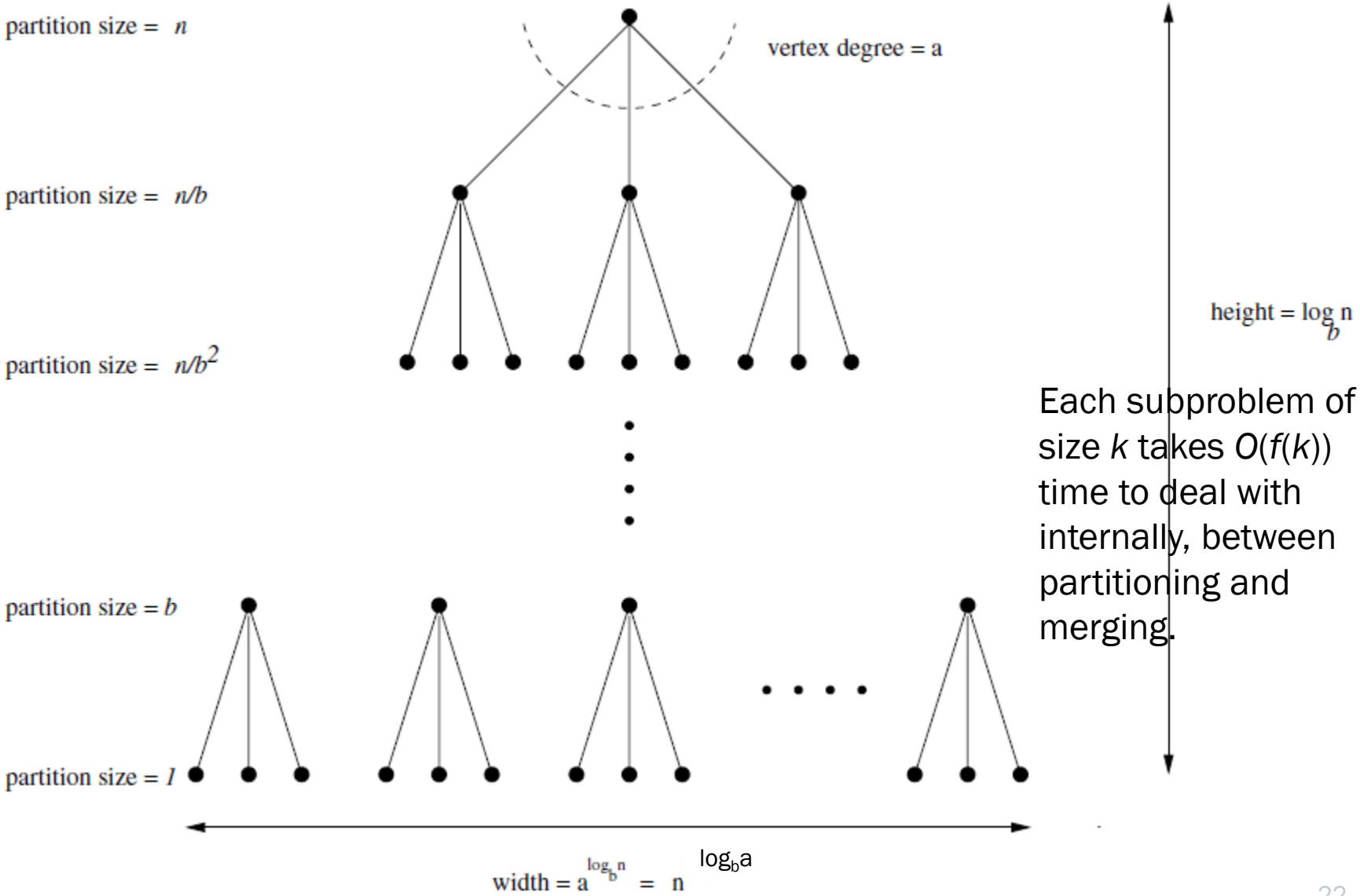
$$T(n) = aT(n/b) + f(n)$$

solutions typically fall into one of three distinct cases:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant > 0, then $T(n) = \theta(n^{\log_b a})$.
   - + heap construction and matrix multiplication
2. If $f(n) = \theta(n^{\log_b a})$, then $T(n) = \theta(n^{\log_b a} \lg n)$.
   - + mergesort and binary search
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some $c < 1$, then $T(n) = \theta(f(n))$
   - + generally arises for clumsier algorithms, where the cost of combining the subproblems dominates everything.

- Case 1 holds for heap construction and matrix multipli cation, while
- Case 2 holds mergesort and binary search.
- Case 3 generally arises for clumsier algorithms, where the cost of combining the subproblems dominates eve rything.

Recursion Tree associated with a typical $T(n) = aT(n/b) + f(n)$ divide-and-conquer algorithm



Each subproblem of size k takes $O(f(k))$ time to deal with internally, between partitioning and merging.

The three cases of the master theorem correspond to three different costs which might be dominant as a function of *a*, *b*, and *f*(*n*) (compare $n^{\log_b a}$ vs *f*(*n*)) .

- *Case 1: Too many leaves ($n^{\log_b a}$ is polynomially larger)*
  - If the number of leaf nodes outweighs the sum of the internal evaluation cost, the total running time is $O(n^{\log_b a})$.
- *Case 2: Equal work per level ($n^{\log_b a}$ and f(n) are same in size)*
  - As we move down the tree, each problem gets smaller but there are more of them to solve. If the sum of the internal evaluation costs at each level are equal, the total running time is the cost per level ($n^{\log_b a}$) times the number of levels ($\log_b a$), for a total running time of $O(n^{\log_b a} \lg n)$.
- *Case 3: Too expensive a root (f(n) is* polynomially larger*)*
  - If the internal evaluation costs grow rapidly enough with *n*, then the cost of the root evaluation may dominate. If so, the the total running time is *O*(*f*(*n*)).