About Programming Assignment #1

- **Readme/script file should include any information/instructions to compile and run your code successfully, for example**
  - Any additional library you used
  - Make file
  - Command lines to compile/run your code
  - Where you put the “input.txt” file …
- **Numbers to sort should be floating-point numbers. You should be able to handle a random number of floating-point numbers, e.g., 1.221, 78.1, and 6661.1112.**
- **For the empirical study, you’d better use logarithmic scale for both horizontal (input size) and vertical (time) axes.**
- **An appropriate time unit is microsecond or at least millisecond**

Quicksort

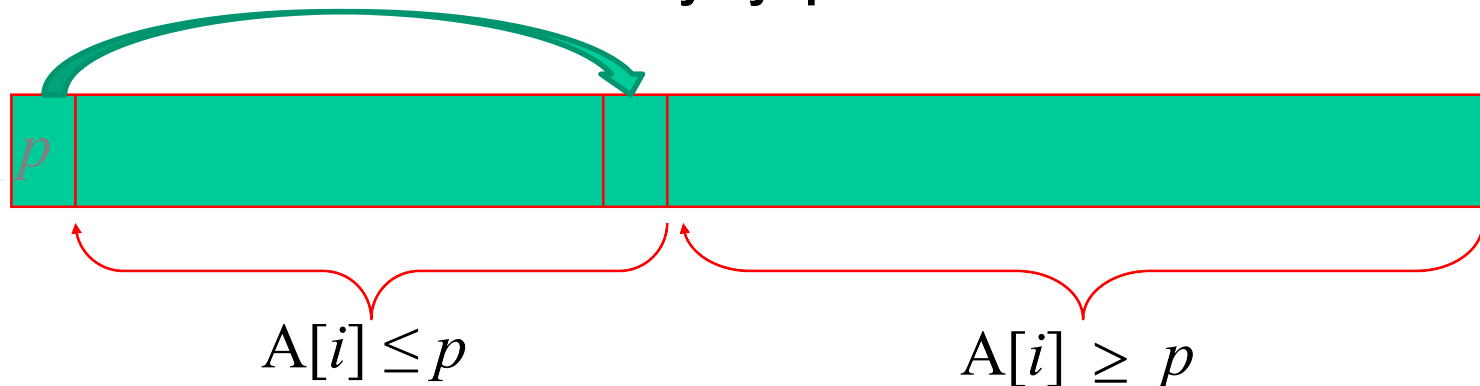
Select a *pivot* (partitioning element)

Rearrange the list so that all the elements in the positions before the pivot are smaller than or equal to the pivot and those after the pivot are larger than or equal to the pivot

Exchange the pivot with the last element in the first (i.e., ≤) sublist– the pivot is now in its final position

Partition into two sublists.

Sort the two sublists individually by quicksort



Efficiency of Quicksort

Basic operation: ***key comparison***

Best case: ***split in the middle — Θ(n log n)***

$$C_{best} = 2C_{best}(\lfloor n/2 \rfloor) + f(n) \qquad \textbf{\textit{for}} \quad n > 1, C_{best}(1) = 0$$

$$f(n) = \begin{cases} n+1 & i \neq j \\ n & i = j \end{cases}$$

⇨ ***So you don't need to count***

Master Theorem: a=2,b=2,k=1

$$C_{best} \in \Theta(n \log n)$$

Efficiency of Quicksort

Worst case: *sorted array! — Θ(n²)*

$$C_{worst}(n) = C_{worst}(n-1) + n + 1$$

Average case: *random arrays — Θ(n log n)*

Assumption: the partition can happen in any position $0 \le p \le n-1$ ***with an equal probability***

$$C_{avg}(0) = 0, C_{avg}(1) = 0$$

$$C_{avg}(n) = \sum_{p=0}^{n-1} \left\{ \frac{1}{n} * \left[(n+1) + C_{avg}(p) + C_{avg}(n-1-p) \right] \right\} \approx 2n \ln n$$

probability — $\frac{1}{n}$; *First subarray* — $C_{avg}(p)$; *second subarray* — $C_{avg}(n-1-p)$

Improvements of Quicksort

- **Better pivot selection: median-of-three partitioning avoids worst case in sorted files**
- **Quicksort is effective for large array**
  - Switch to insertion sort on small subarrays

Possible issue: Not stable!

- Stability: the relative order or records with equal search keys is not changed during sorting

Mergesort vs. Quicksort

	Mergesort	Quicksort
Basic operation	key comparison	key comparison
Best case	O(nlogn)	O(nlogn)
Average case	O(nlogn)	O(nlogn)
Worst case	O(nlogn)	$O(n^2)$
Stable	yes	no
In place	no	yes

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

...

if $B[i] \leq C[j]$

$A[k] \leftarrow B[i]; i \leftarrow i+1$

else $A[k] \leftarrow C[j]; j \leftarrow j+1$

...

Algorithm Partition

repeat

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

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

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

until $i \geq j$

Inner loop procedure

Selection Problem

Find the kth smallest element in A[1],…A[n]. Special cases:

- *minimum:* $k = 1$
- *maximum:* $k = n$
- *median:* $k = \lceil n/2 \rceil$

How can we solve the problem?

Partition-based Selection

pivot/split at A[s] using partitioning algorithm from quicksort

if $s=k$ return A[s]

else if $s<k$ repeat with sublist A[s+1],…A[n].

else if $s>k$ repeat with sublist A[1],…A[s-1].

It is actually a variable size decrease & conquer algorithm. Why?

Efficiency of Partition-based Selection

worst case: $T(n) = T(n-1) + (n+1) \rightarrow \Theta(n^2)$

best case: $\Theta(n)$

average case: $T(n) = T(n/2) + (n+1) \rightarrow \Theta(n)$

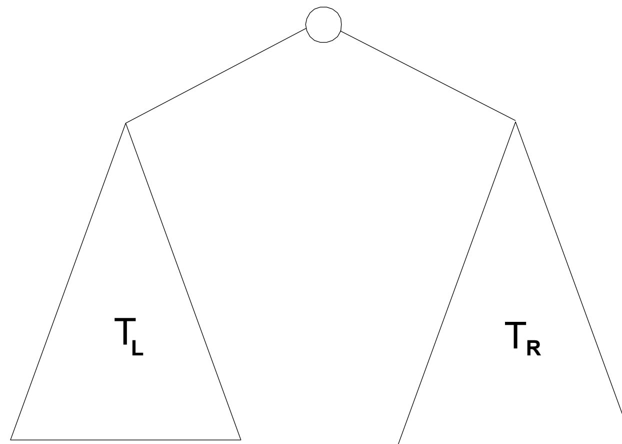
Why we use this algorithm?

Bonus: ***also identifies the k smallest elements (not just the*** k^{th}***)***

Binary Tree Traversals

A binary tree *T* is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees T_L and T_R called the left and right subtree of the root

Solve the subproblem for each subtree by divide-conquer



Height of a binary tree

ALGORITHM $Height(T)$

if $T = \phi$ **return** -1

else return $\max\{Height(T_L), Height(T_r)\} + 1$

Input size n(T): # nodes in T

Basic operation: "+"

Recurrence: $A(n(T)) = A(n(T_L)) + A(n(T_r)) + 1, \quad$ **for** $n(T) > 0$

$$A(0) = 0$$

Basic operation: check if it is an empty tree

Recurrence:

$$C(n(T)) = C(n(T_L)) + C(n(T_r)) + 1, \quad \text{for } n(T) > 0$$

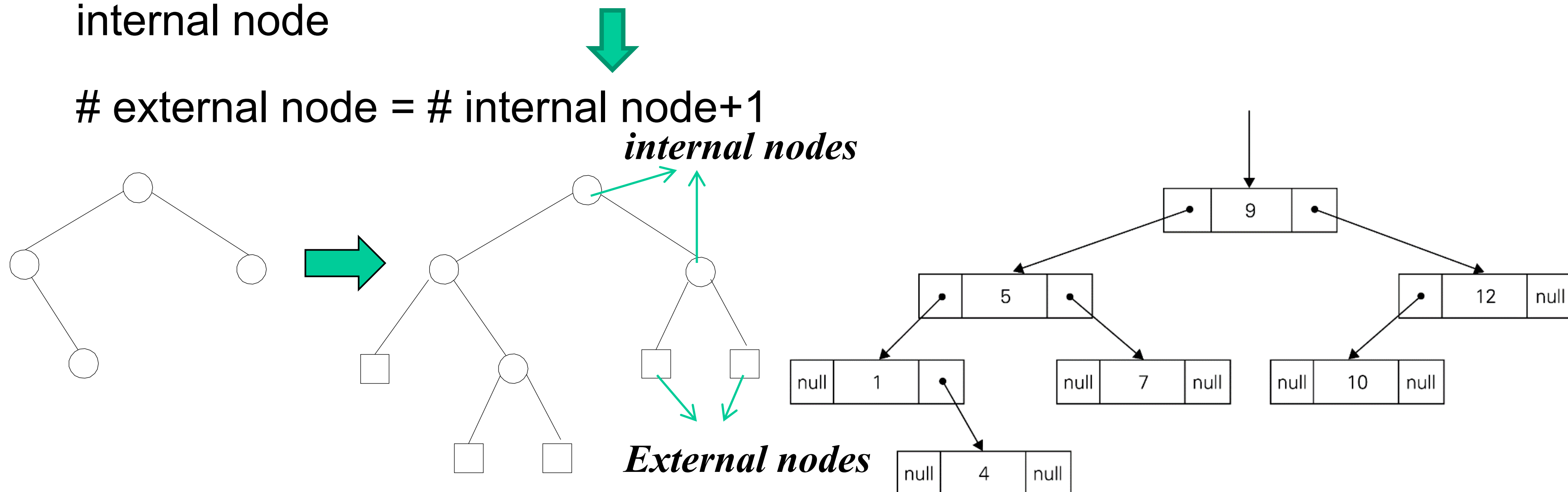
$$C(0) = 1, C(1) = 3$$

Extended Binary Tree

Extended binary tree becomes a full binary tree (strictly binary tree)

- Internal nodes – the original nodes
- External nodes – the special nodes replacing the empty subtrees and empty leaf nodes
- Total number of nodes = 2 * #internal_node+1 = # external node +# internal node

external node = # internal node+1



Height of a binary tree

When basic operation is checking if it is an empty tree, all nodes including external and internal nodes are visited

→C(n) = #external nodes+#internal nodes =2n(T)+1

When basic operation is “+”, only the internal nodes are visited

→ $A(n)$ = #internal nodes = n(T)

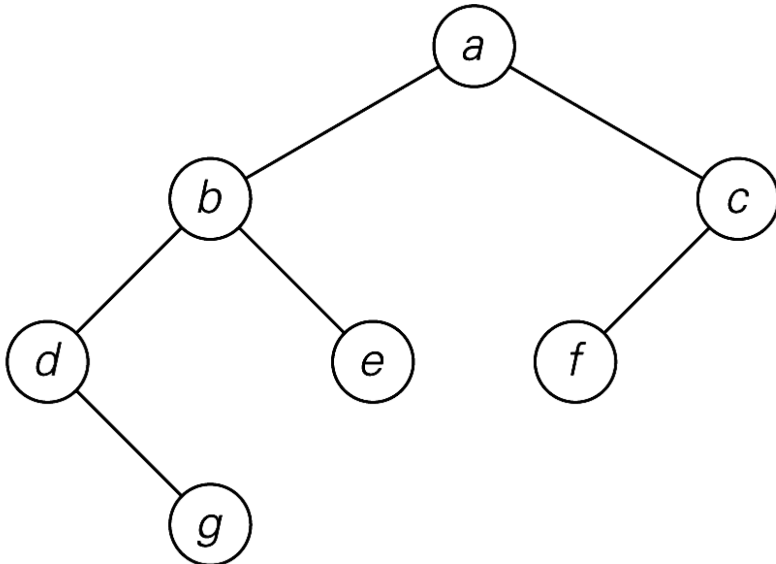
Traverse the binary tree

List all the nodes

Depth-first traversal: visit as far as possible along each subtree

- Preorder traversal: root → left subtree → right subtree
- Inorder traversal: left subtree → root → right subtree
- Postorder traversal: left subtree → right subtree → root

Breadth-first traversal: visit every level from low to right



Preorder: a, b, d, g, e, c, f

Inorder: d, g, b, e, a, f, c

Postorder: g, d, e, b, f, c, a

Breadth-first: a, b, c, d, e, f, g

Traverse the binary tree

Preorder traversal: root → left subtree → right subtree

ALGORITHM $preorder(T)$
if $T \neq \phi$
    output T_{root}
    $preorder(T_L)$
    $preorder(T_r)$

What is the efficiency? $2n+1$

Large Integer Multiplication

Some applications, notably modern cryptology, require manipulation of integers that are over 100 decimal digits long

Such integers are too long to fit a single word of a computer

Therefore, they require special treatment

Consider the multiplication of two such long integers

$$X = x_{n-1}x_{n-2}\cdots x_1 x_0 = \sum_{i=0}^{n-1} x_i r^i$$

Classic paper-and-pencil algorithm

$$Y = y_{n-1}y_{n-2}\cdots y_1 y_0 = \sum_{j=0}^{n-1} y_j r^j$$

***n*²** digit multiplications

$$XY = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} x_i y_j r^{i+j}$$

Large Integer Multiplication – Divide&Conquer

We want to calculate 23 x 14

Since $23 = 2 \cdot 10^1 + 3 \cdot 10^0$ **and** $14 = 1 \cdot 10^1 + 4 \cdot 10^0$

We have

$$23 * 14 = (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0)$$
$$= (2*1)10^2 + (3*1 + 2*4)10^1 + (3*4)10^0$$

Which includes four digit multiplications (*n*²)

But

$$3*1 + 2*4$$
$$= (2+3)*(1+4) - \boxed{(2*1)} - \boxed{(3*4)}$$

Computed already!

Therefore, we only need three digit multiplications

One Formula

Given $a=a_1a_0$ and b=b_1b_0, compute $c=a$*b

We have

$$c = a * b = c_2 10^2 + c_1 10^1 + c_0 10^0$$

where

$$c_2 = a_1 * b_1$$

$$c_0 = a_0 * b_0$$

$$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$$

That means only three digit multiplications are needed to multiply two 2-digit integers

To Multiply Two *n*-digit integers

Assume *n* is even, write

$a = a_1 10^{n/2} + a_0$ **and** $b = b_1 10^{n/2} + b_0$ ***For example, for "1234", a_1= 12, a_0=34, n=4***

Then

$$c = a * b = c_2 10^n + c_1 10^{n/2} + c_0 10^0$$

where

$$c_2 = a_1 * b_1$$

$$c_0 = a_0 * b_0$$

$$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$$

To calculate the involved three multiplications – recursion!
Stops when *n*=1

Efficiency

The recurrence relation is

$$T(n) = 3T(n/2) \textbf{ for } n > 1, T(1) = 1$$

Solving it by backward substitution for n=2^k yields

$$T(2^k) = 3T(2^{k-1}) = 3^2 T(2^{k-2})$$
$$= 3^k T(2^{k-k}) = 3^k$$

Therefore,

$$T(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585} < n^2$$

Reading Assignments

Chapter 5.4 Strassen's Matrix Multiplication

Chapter 5.5 Closest pair and convex-hull by divide-and-conquer