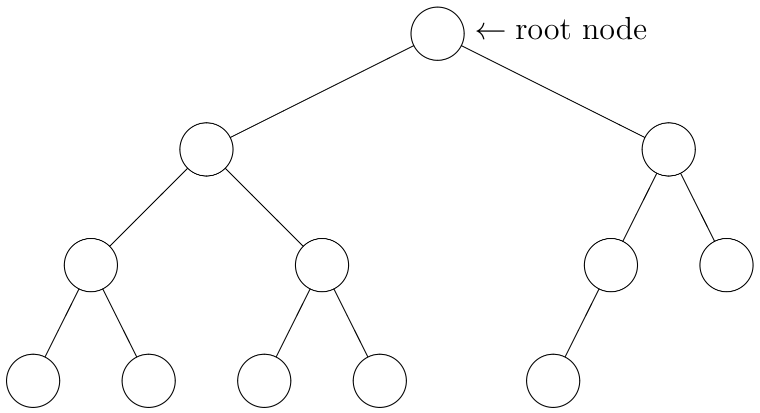
COMP 460 Lecture Notes

R. I. Greenberg

1 Heapsort

- Uses data structure called a *(binary) heap* that can be viewed as a binary tree. (See CLRS 3rd ed. Sec. B.5.3 of for tree terminologies.)



  Each node connects up to two "children" (drawn below it), a left and/or right child. Each node except the root has one parent.

  Each node holds one of the data values to be sorted.

  In a heap, the tree is nearly complete, i.e., each row is full of nodes except that the last row can be empty to the right of some point.

  The heap can actually be represented by an array:

  - root at position 1
  - left child of i at $2i$
  - right child of i at $2i + 1$

  This implies that for $i > 1$, the parent of i is at $\lfloor i/2 \rfloor$.

- HEAPSORT *sorts in place*, i.e., it requires only a constant amount of storage outside of the input array. This was true of insertion sort but not merge sort (w. naive merge).

- Sometimes, only a portion of the array belongs to the heap:
  - length$[A] =$ no. of elts. in array A
  - heap-size$[A] =$ no. of elts. in heap
- **The heap property:**

$$A[\textsc{Parent}(i)] \geq A[i] \ .$$

1.1 A Key Procedure: $\textsc{Heapify}(A, i)$

Given that left and right subtrees below i are heaps, make subtree at i into a heap.

Compare $A[i]$ with its two children. Swap largest of the 3 values into $A[i]$, and if it came from one of the children, step down tree in that direction, and recur.

The running time of HEAPIFY is $O(h)$ for a node of height h.

(The height of a node is the no. of edges on a longest path from the node down to a leaf, and it is $O(\lg n)$ for any node in an n elt. heap.)

1.2 Building a Heap

Make entire array into a heap by applying HEAPIFY in a bottom-up fashion:

```
   BUILD-HEAP($A$)
1  heap-size[$A$] ← length[$A$]
2  for $i$ ← ⌊length[$A$]/2⌋ downto 1
3      HEAPIFY($A, i$)
4  endfor
```

Time of BUILD-HEAP is certainly

$$\leq \frac{n}{2} \cdot O(\lg n) = O(n \lg n)$$

based on the maximum time for HEAPIFY.

Actually, the time is:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil n/2^{h+1} \right\rceil \cdot O(h) \qquad \text{based on time for HEAPIFY on a node of height } h$$

$$\begin{aligned} &\leq O(n \sum_{h=0}^{\infty} h/2^h) \\ &= O(n) \text{ by differentiating a geom. series as in Section A.1 of CLRS 3rd ed.} \end{aligned}$$

1.3 Heapsort Algorithm

Build a heap. Then repeatedly swap first (max.) elt. in heap with last and rebuild a heap that is one elt. smaller.

Rebuild is just one call of Heapify$(A, 1)$ after decrementing heap-size$[A]$.

Heapsort time is $\underbrace{O(n)}_{\text{Build-Heap}} + \underbrace{(n-1)O(\lg n)}_{\substack{n-1 \text{ calls to} \\ \text{Heapify}}} = O(n \lg n)$.

1.4 Priority Queues

In practice, Heapsort is probably not quite the fastest sorting method , but heaps have another application:

A priority queue supports:

- Insert(S, x): Insert elt. x into set S.
- Maximum(S): Return largest elt. of S.
- Extract-Max(S): Remove and return largest elt. of S.
- Increase-key(S, x, k): Increase value of elt. x's key to k

Applications:

- scheduling jobs on a shared computer according to priority
- scheduling events in a simulator
- ...

Jobs (or events) can arise at any time and be added to queue. When ready to service a job (or simulate an event), pull out the one with highest priority (or earliest time of occurrence).

Implementations:

- Maximum: Just read off $A[1]$. Time: $\Theta(1)$.
- Extract-Max: After reading $A[1]$, replace it with last elt. of heap and use Heapify to rebuild heap one size smaller.
  Time: dominated by call of Heapify: $O(\lg n)$.
- Insert: Increase heap size by 1 and put new elt. at end. Then walk up tree to slip it into correct place (like adding an elt. in Insertion-Sort).
  Time: $O(\lg n)$
- Increase-Key: Just increase the key value and walk up as for Insert.

COMP 460 Lecture Notes

R. I. Greenberg

1 Quicksort

1.1 Quicksort (Deterministic Version)

Worst-case time $\Theta(n^2)$ but average-case $\Theta(n \lg n)$ with small constant hidden in the Θ.

```
    QUICKSORT($A, p, r$)        (Initial call is QUICKSORT($A, 1, \text{length}[A]$).)
1   if $p < r$ then
2         $x \leftarrow A[r]$        ($x$ is the "pivot" elt.)
3         Rearrange $A[p..r]$ so that $A[q] = x$ (for some $q$ with $p \le q \le r$),
            and $A[i] \le x \le A[j]$ whenever $i \le q \le j$.
4         QUICKSORT($A, p, q-1$)
5         QUICKSORT($A, q+1, r$)
6   endif
```

No merge is required due to the way we've partitioned.

Step 3 needs some elaboration: Easy to do in $\Theta(n)$ time (when $n = r - p$) using an auxiliary array. Book shows can even do it in place.

1.2 Quicksort Performance

Depends on sizes of subarrays into which we partition.

1.2.1 Worst-Case

We'll see later it's when we divide an n elt. array into arrays of size $n-1$ and 0 (plus the pivot). If we do this at each stage,

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

(Occurs when input already sorted!)

1.2.2 Best-Case

When always divide array in half,

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

1.2.3 Average-Case

It's likely that many splits will be pretty balanced, so we'll see later avg. case is $\Theta(n \lg n)$ with a little larger constant.

2 Randomized Quicksort

- deterministic alg.: For avg. case, we need to assume, e.g., all inputs equally likely. We get in trouble if a bad input is presented often .
- randomized alg.: Design so good performance likely regardless of input, e.g., permute input randomly before applying original quicksort alg. We still end up with $\Theta(n^2)$ worst case, but now it depends on bad luck with the random number generator and not on the input.
- An easier to analyze randomized quicksort: Just like original version, except pick pivot elt. at random from $A[p..r]$.

2.1 Quicksort Analysis (Randomized)

2.1.1 Worst-Case

$$T(n) = \max_{0 \le k \le n-1} [T(k) + T(n-1-k)] + \Theta(n) \ .$$

Substitution method: Try $T(n) \le cn^2$:

$$T(n) \le \max_{0 \le k \le n-1} [ck^2 + c(n-1-k)^2] + \Theta(n) \ .$$

The max. is achieved at $k = 0$ or $k = n-1$ (by a calculus argument).

So

$$\begin{aligned} T(n) &\le c(n-1)^2 + \Theta(n) \\ &= cn^2 - 2cn + c + \Theta(n) \\ &\le cn^2 \text{ for large enough constant } c \end{aligned}$$

2.1.2 Average-Case

(Analysis here similar to CLRS 3rd ed. exercise 7–3.)

Like last recurrence, but instead of choosing k to maximize running time, each value of k is roughly equally likely. (Can make partition work this way even if not all elements distinct.)

Then, avg. time (expectation of running time) is

$$\begin{aligned} T(n) &= \Theta(n) + \frac{1}{n}\sum_{k=0}^{n-1}[T(k) + T(n-1-k)] \\ &= \Theta(n) + \frac{2}{n}\sum_{k=0}^{n-1} T(k) \end{aligned}$$

Solve by substitution method: Try $T(n) \leq anlgn$:

$$\begin{aligned} T(n) &= \Theta(n) + \frac{2}{n}\sum_{k=1}^{n-1} T(k) \\ &\leq \Theta(n) + \frac{2}{n}\sum_{k=1}^{n-1} aklgk \\ &= \Theta(n) + \frac{2a}{n}\sum_{k=1}^{n-1} klgk \\ &\leq \Theta(n) + \frac{2a}{n}(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2) \text{ by CLRS 3rd ed. Exercise 7–3d} \\ &= \Theta(n) + an\lg n - \frac{a}{4}n \\ &\leq an \lg n \text{ for large enough constant } a \end{aligned}$$

To handle the base case, you can change the guess to $an \lg n + b$ or prove only for $n \geq n_0$ with $n_0 > 1$.

Here is the proof for CLRS 3rd ed. Exercise 7–3d, using the techniques of "splitting the sum" and "bounding the terms" (which comes up again in the lower bound for comparison-based sorting):

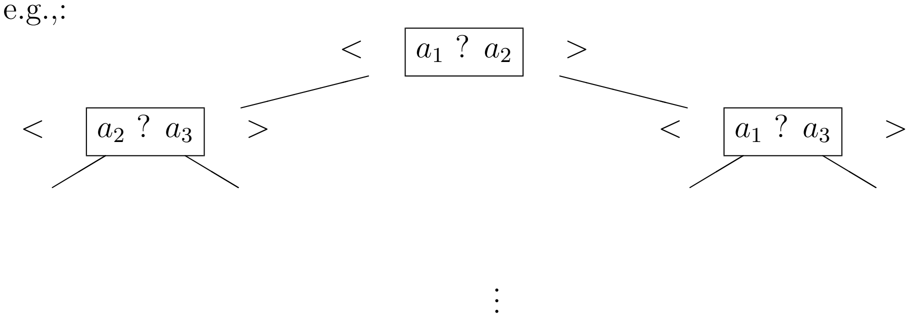
$$\begin{aligned} \sum_{k=1}^{n-1} k \lg k &= \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k \quad + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k \\ &\leq (\lg n - 1)\Big(\sum_{k=1}^{\lceil n/2 \rceil - 1} k\Big) \quad + \quad (\lg n)\Big(\sum_{k=\lceil n/2 \rceil}^{n-1} k\Big) \\ &= (\lg n)\sum_{k=1}^{n-1} k \quad - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \end{aligned}$$

$$\begin{aligned} &\le (\lg n)\frac{n}{2}(n-1) \quad - \quad \frac{n}{4}(\frac{n}{2}-1) \\ &\le \frac{1}{2}n^2 \lg n \quad - \quad \frac{1}{8}n^2 \end{aligned}$$

3 Lower Bound on Comparison Sorts

Worst-case lower bound for algs. that don't use values of elts. except to compare them.

WLOG, assume all elts. distinct; a comparison asks which of two elts. is larger, e.g., $a_i?a_j$. The program may branch according to the answer ($<$ or $>$); described by a "decision tree", e.g.,:



Each execution of the program follows a path from the root to a leaf based on the results of the comparisons. When we reach a leaf, we must have a decision on the sorted order. Thus we need a distinct leaf for each of the $n!$ possible answers.

So $n! \le 2^h$ (where h is the tree height), which implies

$$h \ge \lg(n!) = \Theta(n \lg n)$$

(by CLRS 3rd ed. Exercise 3.2–3)

So the worst-case time is $\Omega(n \lg n)$.

COMP 460 Lecture Notes

R. I. Greenberg

1 Sorting in Linear Time

1.1 Counting Sort

Assumes input elts. $A[1..n]$ are integers in the range 1 to k.

1. $C[i] \leftarrow$ no. of elts. of A that equal i.
   (Do it by marching through input array once and tallying values.)
2. Change $C[i]$ to no. of elts. $\leq i$.
   ($C[i] \leftarrow C[i] + C[i-1]$ for $i = 2, 3, 4, \ldots k$.)
3. Put each elt. of A into correct position of output array B:

```
for $j \leftarrow n$ downto 1
    $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$
endfor
```

This sort is stable: inputs with same value appear in output array in same order as in input array. (This matters when "satellite data" being carried around with keys being sorted.)

Running time: $O(k+n)$
$= O(n)$ when $k = O(n)$.

1.2 Radix Sort

Sorts nos. digit by digit (or other keys field by field).

Let $d =$ no. of digits in a number
$k =$ max. range of digits (e.g., digits from 1 to k or 0 to $k-1$)

k is called the radix or number base.

Intuitive digit-by-digit approach would be:
Sort by most significant digit; then recursively sort the k collections of nos. having same first digit. Gives a lot of subarrays to keep track of.

Radix sort counterintuitively sorts on least significant digit first:

```
   RADIX-SORT($A, d$)
1  for $i \leftarrow 1$ to $d$
2      Do a stable sort of array $A$ on $i$th digit from right.
3  endfor
```

If k not too large, counting sort is a good choice for line 2.

Running time: $\Theta(n+k)$ for each execution of line 2.
Total time: $\Theta(d(n+k))$.

When d constant and $k = O(n)$, time for radix sort is $O(n)$. This is often an appropriate perspective, since we tend to build computers with around $\Theta(\lg n)$-bit numbers for the largest n used in practice.

So radix sort time may be good in practice, but it does not sort in place (when based on counting sort).

1.3 Bucket Sort

$\Theta(n)$ average time for random inputs uniformly distributed over interval $[0, 1)$, for example.

Divide $[0, 1)$ into n equal-sized buckets. Put each elt. in correct bucket. Go through buckets in order, sorting nos. in each bucket by, e.g., insertion sort.

Let n_i be no. of elts. in bucket i for $0 \leq i \leq n-1$. Expected time to do the insertion sorts is

$$\sum_{i=0}^{n-1} \mathrm{E}[O(n_i^2)] = O(\sum_{i=0}^{n-1} \mathrm{E}[n_i^2]) ,$$

which the book shows is $O(n)$. The book gives a somewhat lengthy argument using only elementary principles. Here is a shorter argument using some more advanced results that are derived elsewhere in the text. Letting $p = 1/n$ denote the probability of a particular item following into a particular bucket, we have:

$$\begin{aligned}
\mathrm{E}[n_i^2] &= \mathrm{Var}[n_i] + \mathrm{E}^2[n_i] && \text{By Eqn. C.27} \\
&= np(1-p) + (np)^2 && \text{By Eqns. C.37 and C.39} \\
&= 1 - \frac{1}{n} + 1^2 \\
&= 2 - \frac{1}{n} \\
&= \Theta(1) .
\end{aligned}$$