CS 240E
Personal Notes

Marcus Chan

Taught by Therese Biedl

UW CS '25

Chapter 1: Algorithm Analysis

How to "solve" a problem

When solving a problem, we should

① Write down exactly what the problem is;

eg Sorting Problem
→ given n numbers in an array, put them in sorted order

② Describe the idea;

eg Insertion Sort

Idea: repeatedly move one item into the correct space of the sorted part.

③ Give a detailed description; usually pseudocode.

eg Insertion Sort:

```
for i=1, ..., n-1
    j=i
    while j>0 and A[j-1]>A[j]
        swap A[j] and A[j-1]
        j--
```

④ Argue the correctness of the algorithm.
→ In particular, try to point out loop invariants & variants.

⑤ Argue the run-time of the program.
→ We want a theoretical bound.
(using asymptotic notation).

To do this, we count the # of primitive operations.

Primitive operations

In our computer model,
① our computer has memory cells
② all cells are equal
③ all cells are big enough to store our numbers

Then, "primitive operations" are $+, -, \times, \div$, load & following references.

We also assume each primitive operation takes the same amount of time to run.

Asymptotic notation

Big-O notation: $O(f(n))$

We say that "$f(n) \in O(g(n))$" if there exist $c>0$, $n_0>0$ s.t.

$$|f(n)| \le c|g(n)| \quad \forall n \ge n_0.$$

eg $f(n) = 7sn + 500$ & $g(n) = 5n^2$, $c=1$ & $n_0 = 20$

Usually, "n" represents input size.

Show $2n^2 + 3n^2 + 11 \in O(n^2)$

To show the above, we need to find c, n_0 such that $0 \le 2n^2+3n+11 \le cn^2 \quad \forall n \ge n_0$.

Sol^n. Consider $n_0 = 1$. Then

$1 \le n \Rightarrow 1 \le n^2 \Rightarrow 11 \le 11n^2$

$1 \le n \Rightarrow n \le n^2 \Rightarrow 3n \le 3n^2$

$2n^2 \le 2n^2$ (+)

$\Rightarrow 2n^2+3n+11 \le 11n^2+2n^2+3n^2 = 16n^2$

Hence $c=16$ & $n_0=1$, so $2n^2+3n+11 \in O(n^2)$. ☒

Ω-notation (Big Omega): $f(n) \in \Omega(g(n))$

We say "$f(n) \in \Omega(g(n))$" if there exist $c>0$, $n_0>0$ such that

$$c|g(n)| \le |f(n)| \quad \forall n \ge n_0.$$

Θ-notation (Big Theta): $f(n) \in \Theta(g(n))$

We say "$f(n) \in \Theta(g(n))$" if there exist $c_1, c_2 > 0$, $n_0 > 0$ such that

$$c_1|g(n)| \le |f(n)| \le c_2|g(n)|.$$

Note that

$$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \ \&\ f(n) \in \Omega(g(n)).$$

o-notation (small o): $f(n) \in o(g(n))$

We say "$f(n) \in o(g(n))$" if for any $c>0$, there exists some $n_0>0$ such that

$$|f(n)| < c|g(n)| \quad \forall n \ge n_0.$$

If $f(n) \in o(g(n))$, we say $f(n)$ is "asymptotically strictly smaller" than $g(n)$.

ω-notation (small omega): $f(n) \in \omega(g(n))$

We say $f(n) \in \omega(g(n))$ if for all $c>0$, there exists some $n_0>0$ such that

$$0 \le c|g(n)| < |f(n)| \quad \forall n \ge n_0.$$

FINDING RUNTIME OF A PROGRAM

To evaluate the run-time of a program, given its pseudocode, we do the following:

① Annotate any primitive operations with just "$\Theta(1)$";

② For any loops, find the worst-case bound for how many times it will execute;

③ Calculate the big-O run time of the program;

④ Argue this bound is tight (ie show program is also in $\Omega(g(n))$, so runtime $\in \Theta(g(n))$.)

eg insertion sort

```
for i=1, ..., n-1                          Θ(1)
    j=i                                    Θ(1)
    while j>0 and A[j-1]>A[j]              Θ(1)   } Θ(n)   } Θ(n)
        swap A[j] and A[j-1]               Θ(1)   }
        j--                                Θ(1)
```

Then, let c be a const s.t. the upper bounds all the times needed to execute one line.

So runtime $\leq n \cdot n \cdot c = c \cdot n^2 \in O(n^2)$.

Next, consider the worst pos. case of insertion sort.

For each $A[i]$, we need $i-1$ swaps.

So

$$\text{runtime} \geq \sum_{i=1}^{n-1} (i-1) = \sum_{i=1}^{n-2} i = \frac{(n-2)(n-1)}{2} \in \Omega(n^2),$$

and so

$$\text{runtime} \in \Theta(n^2).$$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = L < \infty \Rightarrow f(n) \in O(g(n))$

<< LIMIT RULE I >> (L1.1(2))

Let $f(n), g(n)$ be such that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = L < \infty$.

Then necessarily $f(n) \in O(g(n))$.

Proof. We know $\lim_{n\to\infty} \frac{f(n)}{g(n)} = L$.

$\Rightarrow \forall \varepsilon > 0: \exists n_\varepsilon$ s.t. $\left|\frac{f(n)}{g(n)} - L\right| < \varepsilon \quad \forall n \geq n_\varepsilon$.

We want to show

$\exists c > 0, \exists n_0$ s $\forall n \geq n_0$, $f(n) \leq c \cdot g(n)$.

Choose $\varepsilon = 1$. Then there exists a n_1 s.t.

$\forall n \geq n_1$, $\left|\frac{f(n)}{g(n)} - L\right| \leq 1$.

$\Leftrightarrow \frac{f(n)}{g(n)} - L \leq \left|\frac{f(n)}{g(n)} - L\right| \leq 1$.

$\Leftrightarrow \frac{f(n)}{g(n)} \leq L + 1$

$\Leftrightarrow f(n) \leq L g(n) + g(n)$ (since $g(n) > 0$)

Choose $c = L + 1$. Note $f(n), g(n) > 0$, so $L + 1 > 0$, and since $L < \infty$, thus $c < \infty$.

Now, for all $n \geq n_1$, $f(n) \leq c \cdot g(n)$, and so $f(n) \in O(g(n))$. ∎

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Leftrightarrow f(n) \in o(g(n))$

<< LIMIT RULE II >> (L1.1(1))

Let $f(n), g(n)$. Then $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ iff $f(n) \in o(g(n))$.

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = L > 0 \Rightarrow f(n) \in \Omega(g(n))$

<< LIMIT RULE III >> (L1.1(3))

Let $f(n), g(n)$ such that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = L > 0$.

Then necessarily $f(n) \in \Omega(g(n))$.

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) \in \omega(g(n))$

<< LIMIT RULE IV >> (L1.1(4))

Let $f(n), g(n)$ such that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Then necessarily $f(n) \in \omega(g(n))$.

OTHER LIMIT RULES

The following are corollaries of the limit rules:

① $f(n) \in \Theta(f(n))$ } (Identity)

② $K \cdot f(n) \in \Theta(f(n)) \quad \forall K \in \mathbb{R}$ } (Constant multiplication)

③ $f(n) \in O(g(n))$, $g(n) \in O(h(n))$ $\Rightarrow f(n) \in O(h(n))$

④ $f(n) \in \Omega(g(n))$, $g(n) \in \Omega(h(n))$ $\Rightarrow f(n) \in \Omega(h(n))$ } (Transitivity)

⑤ $f(n) \in O(g(n))$, $g(n) \leq h(n) \ \forall n \geq N$ $\Rightarrow f(n) \in O(h(n))$

⑥ $f(n) \in \Omega(g(n))$, $g(n) \geq h(n) \ \forall n \geq N$ $\Rightarrow f(n) \in \Omega(h(n))$ } (Dominance)

⑦ $f_1(n) \in O(g_1(n))$, $f_2(n) \in O(g_2(n))$ $\Rightarrow f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$

⑧ $f_1(n) \in \Omega(g_1(n))$, $f_2 \in \Omega(g_2(n))$ $\Rightarrow f_1(n) + f_2(n) \in \Omega(g_1(n) + g_2(n))$ } (Addition)

⑨ $h(n) \in O(f(n) + g(n))$ $\Rightarrow h(n) \in O(\max(f(n), g(n)))$ } (Maximum-rule for O)

⑩ $h(n) \in \Omega(f(n) + g(n))$ $\Rightarrow h(n) \in \Omega(\max(f(n), g(n)))$ } (Maximum-rule for Ω)

$f(n) \in P_d(\mathbb{R}) \Rightarrow f(n) \in \Theta(n^d)$ << POLYNOMIAL RULE >>

Let $f(n) \in P_d(\mathbb{R})$, ie of the form $f(n) = c_0 + c_1 n + \cdots + c_d n^d$.

Then necessarily $f(n) \in \Theta(n^d)$.

$b > 1$; $\log_b(n) \in \Theta(\log n)$ << LOG RULE I >>

Let $b > 1$. Then necessarily $\log_b(n) \in \Theta(\log n)$.

*convention: "log" = "$\log_2$".

Proof. Note

$$\lim_{n\to\infty} \frac{\log_b(n)}{\log n} = \lim_{n\to\infty} \frac{\left(\frac{\log(n)}{\log(b)}\right)}{\log(n)} = \lim_{n\to\infty} \frac{1}{\log(b)} > 0,$$

So by Limit Rules 1 & 3, $\log_b(n) \in \Theta(\log n)$. ∎

$c, d > 0$; $\log^c n \in o(n^d)$ << LOG RULE II >>

Let $c, d > 0$. Then necessarily $\log^c n \in o(n^d)$,

where $\log^c n \equiv (\log n)^c$.

Proof. See that

$$\lim_{n\to\infty} \frac{\ln^k n}{n} \overset{L'H}{=} \lim_{n\to\infty} \frac{k \ln^{k-1}(n) \cdot \frac{1}{n}}{1} \overset{L'H}{=} \cdots \overset{L'H}{=} \lim_{n\to\infty} \frac{k!}{n} = 0,$$

so $\ln^k n \in o(n)$.

Fix $c, d > 0$. Then

$$\lim_{n\to\infty} \frac{\ln^c n}{n^d} = \left(\lim_{n\to\infty} \frac{\ln^{\frac{c}{d}} n}{n}\right)^d \leq \left(\lim_{n\to\infty} \frac{\ln^{\lceil \frac{c}{d} \rceil} n}{n}\right)^d = 0^d = 0.$$

As $\log^c n = \left(\frac{1}{\ln 2}\right)^c \ln^c n$, thus $\lim_{n\to\infty} \frac{\log^c n}{n^d} = \left(\frac{1}{\ln 2}\right)^c \cdot 0 = 0$.

Proof follows from the limit rule. ∎

$f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$

Suppose $f(n) \in o(g(n))$. Then $f(n) \in O(g(n))$.

$f(n) \in o(g(n)) \Rightarrow f(n) \notin \Omega(g(n))$

Suppose $f(n) \in o(g(n))$. Then $f(n) \in \Omega(g(n))$.

Proof. Prove by contrative:

$f(n) \in \Omega(g(n)) \Rightarrow f(n) \notin o(g(n))$.

Consider cases for $\lim_{n\to\infty} \frac{f(n)}{g(n)}$.

Case 1 It DNE.

$\Rightarrow f(n) \notin o(g(n))$.

Case 2 Limit exists.

Then by $f(n) \in \Omega(g(n))$, thus

$f(n) \geq c \cdot g(n)$ for some $c > 0$ & $n \geq n_0$.

$\Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c > 0$

$\Rightarrow$ limit $\neq 0$

$\Rightarrow f(n) \notin o(g(n))$. ∎

$f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$

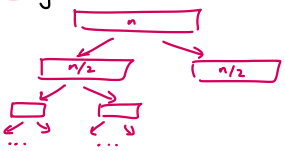
Suppose $f(n) \in \omega(g(n))$. Then $f(n) \in \Omega(g(n))$.

WORST-CASE RUNTIME: $T_A^{worst}(n)$

The "worst-case runtime" for an algorithm, denoted $T_A^{worst}(n)$, is the max run-time among all instances of size n.

ANALYZING RECURSIVE ALGORITHMS

Consider merge sort:



Analysis of MergeSort:

```
MergeSort(A, n, ℓ←0, r←n-1, S←NIL)
A: array of size n,   0≤ℓ≤r≤n-1
    if S is NIL   init it as array S[0...n-1]
    if (r≤ℓ) then
        return
    else
        m = (r+ℓ)/2
        MergeSort(A, n, ℓ, m, S)
        MergeSort(A, n, m+1, r, S)
        Merge(A, ℓ, m, r, S)

Merge(A, ℓ, m, r, S)
A[0,...,n-1] is an array.  A[ℓ,...,m] is sorted,
A[m+1,...,r] is sorted,  S[0,...,n-1] is an array
    copy A[ℓ...r] into S[ℓ...r]
    int i_L ← ℓ ;  int i_R ← m+1
    for (k←ℓ; k≤r; k++) do
        if (i_L > m)  A[k] ← S[i_R++]
        else if (i_R > r)  A[k] ← S[i_L++]
        else if (S[i_L] ≤ S[i_R])  A[k] ← S[i_L++]
        else  A[k] ← S[i_R++]
```

Arguing run-time:

Let $T(n)$ = run-time of merge sort with n items.

$$\therefore T(n) \leq \begin{cases} c, & n \leq 1 \\ 2 \cdot T(\frac{n}{2}) + c \cdot n, & \text{otherwise} \end{cases}$$

$\in \Theta(n \log n)$ (see below)

SOME RECURRENCE RELATIONS

Note:

Recursion...	... resolves to ...	Example
$T(n) = T(n/2) + \Theta(1)$	$T(n) \in \Theta(\log n)$	Binary search
$T(n) = 2T(n/2) + \Theta(n)$	$T(n) \in \Theta(n \log n)$	Mergesort
$T(n) = 2T(n/2) + \Theta(\log n)$	$T(n) \in \Theta(n)$	Heapify
$T(n) = T(cn) + \Theta(n),\ 0 < c < 1$	$T(n) \in \Theta(n)$	Selection
$T(n) = 2T(n/4) + \Theta(1)$	$T(n) \in \Theta(\sqrt{n})$	Range Search
$T(n) = T(\sqrt{n}) + \Theta(1)$	$T(n) \in \Theta(\log \log n)$	Interpolation Search

SORTING PERMUTATION [OF AN ARRAY]

A "sorting permutation" of an array A is the permutation $\pi : \{0, \dots, n-1\} \to \{0, \dots, n-1\}$ such that

$$A[\pi(0)] \leq A[\pi(1)] \leq \dots \leq A[\pi(n-1)].$$

* $A \circ \pi$ would be "sorted".

eg if $A = [14, 3, 2, 6, 1, 11, 7]$, then $\pi = \{4, 2, 1, 3, 6, 5, 0\}$.

For a sorting permutation π, we define its inverse π^{-1} to be the array which entries have exactly the same "relative order" as in A.

eg $\pi^{-1} = [6, 2, 1, 3, 0, 5, 4]$ (if A is as above)

We denote "Π_n" to be the set of all sorting permutations of $\{0, \dots, n-1\}$.

AVERAGE-CASE RUNTIME: $T_A^{avg}(n)$

The "average-case run-time" of an algorithm is defined to be

$$T_A^{avg}(n) := \underset{I \in \mathcal{I}_n}{avg}\ T_A(I) = \frac{1}{|\mathcal{I}_n|} \sum_{I \in \mathcal{I}_n} T_A(I),$$

where $\mathcal{I}_n$ is the set of instances of size n.

In particular, if we can map each instance I to a permutation $\pi \in \Pi_n$, where Π_n is the set of all permutations of $\{0, \dots, n-1\}$, then we may alternatively state that

$$T^{avg}(n) = \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi)$$

where $T(\pi)$ is the number of comparisons on instance π^{-1}.

EXAMPLE: AVGCASEDEMO

Consider the algorithm

```
avgCaseDemo(A,n)
// array A stores n distinct numbers
1. if n≤2 return
2. if A[n-2] ≤ A[n-1] then
3.     avgCaseDemo(A[0...⌈n/2⌉-1], ⌈n/2⌉)   // good case
4. else
5.     avgCaseDemo(A[0...n-3], n-2)          // bad case
```

We claim $T^{avg}(n) \in O(\log n)$.

Proof. To avoid constants, let $T(\cdot) :=$ # of recursions; the run-time is proportional to this.

As all numbers are distinct, we may associate each array with a sorting permutation.

So for $\pi \in \Pi_n$, let $T(\pi)$ = # of recursions done if the input array has sorting permutation π.

Note we have two kinds of permutations:

① "Good" permutations — if $A[n-2] < A[n-1]$; or

② "Bad" permutations — if $A[n-2] > A[n-1]$.

Denote

Π_n^{good} = # of good permutations of size n

& Π_n^{bad} = # of bad permutations of size n.

Then, we claim that

$$\sum_{\pi \in \Pi_n^{good}} T(\pi) \le |\Pi_n^{good}|\,(1+T^{avg}(\lfloor \tfrac{n}{2} \rfloor))$$

$$\& \sum_{\pi \in \Pi_n^{bad}} T(\pi) \le |\Pi_n^{bad}|\,(1+T^{avg}(n-2)).$$

Proof. We only prove this for good permutations; the other claim is similar.

Fix $\pi \in \Pi_n^{good}$, and let π_{half} be the permutation of the recursion;

ie π_{half} = the sorting perm of $\pi^{-1}[0,\dots,\lceil \tfrac{n}{2} \rceil - 1]$ &

$T(\pi) = 1 + T(\pi_{half})$.

Note that $\pi_{half} \in \Pi_{\lfloor n/2 \rfloor}$. Then see that

$$\sum_{\pi \in \Pi_n^{good}} T(\pi) = \sum_{\pi \in \Pi_n^{good}} (1 + T(\pi_{half}))$$

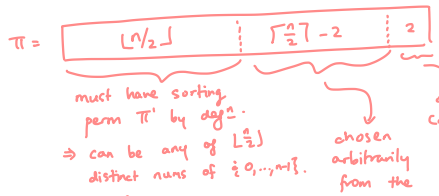
$$= |\Pi_n^{good}| + \sum_{\pi \in \Pi_n^{good}} T(\pi_{half})$$

$$= |\Pi_n^{good}| + \sum_{\pi' \in \Pi_{\lfloor n/2 \rfloor}} |\{\pi \in \Pi_n^{good} \text{ for which } \pi_{half} = \pi'\}| \cdot T(\pi').$$

We next prove the following claim:

Claim If $n \ge 3$, then $|\Pi_n^{good}(\pi')| = \dfrac{n!}{2!(\lfloor \frac{n}{2} \rfloor)!}$ $\forall \pi' \in \Pi_{\lfloor n/2 \rfloor}$.

Proof. Fix $\pi \in \Pi_n^{good}(\pi')$. See that



must have sorting perm π' by def^n.
⇒ can be any of $\lfloor \frac{n}{2} \rfloor$ distinct nums of $\{0,\dots,n-1\}$.
⇒ $\binom{n}{\lfloor n/2 \rfloor}$ choices.

chosen arbitrarily from the remaining nums
⇒ $\binom{\lceil n/2 \rceil}{\lceil n/2 \rceil - 2}$ choices

one choice (order predetermined ∵ this is a good perm)

Hence, the # of choices is

$$\binom{n}{\lfloor n/2 \rfloor} \cdot \binom{\lceil n/2 \rceil}{\lceil n/2 \rceil - 2} (\lceil n/2 \rceil - 2)! = \cdots = \frac{n!}{\lfloor n/2 \rfloor ! \cdot 2},$$

as needed. ※

Then, since $|\Pi_n^{good}| = \frac{1}{2}|\Pi_n| = \frac{n!}{2}$, it follows that $|\Pi_n^{good}| / |\Pi_{\lfloor n/2 \rfloor}|$, and so

$$\sum_{\pi \in \Pi_n^{good}} T(\pi) = |\Pi_n^{good}| + \sum_{\pi' \in \Pi_{\lfloor n/2 \rfloor}} |\Pi_n^{good}(\pi')| \cdot T(\pi')$$

$$= |\Pi_n^{good}| + \sum_{\pi' \in \Pi_{\lfloor n/2 \rfloor}} \frac{|\Pi_n^{good}|}{|\Pi_{\lfloor n/2 \rfloor}|} \cdot T(\pi')$$

$$= |\Pi_n^{good}| \left(1 + \frac{1}{|\Pi_{\lfloor n/2 \rfloor}|} \sum_{\pi' \in \Pi_{\lfloor n/2 \rfloor}} T(\pi')\right)$$

$$= |\Pi_n^{good}| \,(1 + T^{avg}(\lfloor \tfrac{n}{2} \rfloor))$$

as needed. ※

Next, see that $|\Pi_n^{good}| = |\Pi_n^{bad}|$, since we can map any bad perm to a good one (and v.v.) by swapping $A[n-2]$ & $A[n-1]$.

Thus $|\Pi_n^{good}| = |\Pi_n^{bad}| = \frac{n!}{2}$, and so

$$T^{avg}(n) \le \frac{1}{|\Pi_n|} \sum_{I \in \Pi_n} T(I) \le \frac{1}{|\Pi_n|} \left(\sum_{I \in \Pi_n^{good}} T(I) + \sum_{I \in \Pi_n^{bad}} T(I) \right)$$

$$\le \frac{1}{|\Pi_n|} \left(|\Pi_n^{good}| (1 + T^{avg}(\lfloor \tfrac{n}{2} \rfloor)) + |\Pi_n^{bad}| (1 + T^{avg}(n-2)) \right)$$

$$= 1 + \tfrac{1}{2} T^{avg}(\lfloor \tfrac{n}{2} \rfloor) + \tfrac{1}{2} T^{avg}(n-2).$$

Finally, we show $T^{avg}(n) \le 2 \log n$ by induction.

Clearly, this holds for $n \le 2$.

So, assume $n \ge 3$. Assume the inductive hypothesis. Then see that

$$T^{avg}(n) \le 1 + \tfrac{1}{2} T^{avg}(\lfloor \tfrac{n}{2} \rfloor) + \tfrac{1}{2} T^{avg}(n-2)$$

$$\le 1 + \tfrac{1}{2}(2 \log(\lfloor \tfrac{n}{2} \rfloor)) + \tfrac{1}{2}(2 \log(n-2))$$

$$\le 1 + \log(n-1) + \log(n)$$

$$= 2 \log(n),$$

which suffices to prove the claim. ∎

EXPECTED-CASE RUNTIME: $T_A^{exp}(n)$

The "expected-case runtime" of an algorithm is defined to be

$$T_A^{exp}(I) = E[T_A(I,R)] = \sum_{\text{all } R} T_A(I,R)\, P(R)$$

where R is a sequence of random outcomes, and $P(R)$ denotes the probability the random variables in the algorithm A have outcomes R.

Chapter 2: Priority Queues and Heaps

ADT PRIORITY QUEUES

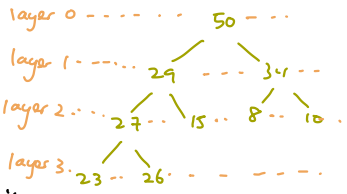
1. A "priority queue" stores items that have a priority, or key.
2. Operations:
   1. insert (a given item & priority as a k-v pair)
   2. deleteMax (return item with largest priority)
   3. size, isEmpty
3. We can use a PQ to sort:

```
PQ-Sort (A[0, ..., n-1])
    init PQ to an empty PQ
    for i←0 to n-1 do
        PQ.insert(A[i])
    for i←n-1 down to 0 do
        A[i] ← PQ.deleteMax()
```

4. The run-time of the above algorithm is O(initialization + n·insert + n·deleteMax).

BINARY HEAPS

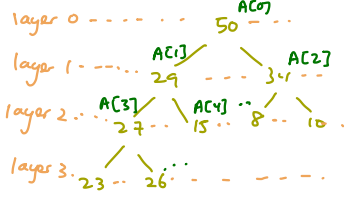
1. "Binary heaps" are binary trees with the following properties:
   1. Each level is filled except the last, which is filled from the left; } "structural" property
   2. key(i) ≤ key(parent(i)) } "heap-order property"



Each node is "shorthand" for a reference to a <key-value> pair:

"50" ⟷ priority = 50, data = <...>

2. Note that binary heaps have height O(log n).
3. We store binary heaps using an array:



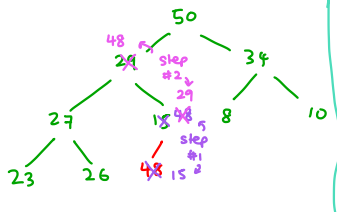
⇒

A	0	1	2	3	4	5	...
	50	29	34	27	15	8	...

Then $parent(i) = \lfloor \frac{i-1}{2} \rfloor$, similar implementation for left/right child.

INSERT IN HEAPS

1. To insert into a heap, we just place the new key at the first free leaf.
2. We also employ "fix-up":

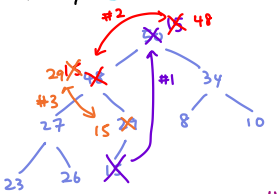


Pseudocode

```
fix-up(A,k)
k: an index corresponding to a node of the heap
    while parent(k) exists and A[parent(k)] < A[k] do
        swap A[k] & A[parent(k)]
        k ← parent(k)
```

DELETEMAX IN HEAPS

1. The maximum item of a heap is just the root node.
2. We replace the root by the last leaf, which is taken out.
3. Then, we use "fix-down":



* we "swap down" from the root-node until the heap-ordering is satisfied.

Pseudocode

```
fix-down(A,n,k)
// A: an array that stores a heap of size n
// k: an index corresponding to a node of the heap
while k is not a leaf do
    // find child with larger key
    j ← left child of k
    if (j is not last(n) & A[j+1] > A[j])
        j ← j+1
    if A[k] ≥ A[j] break
    swap A[j] and A[k]
    k ← j
```

4. Run-time: $O(\text{height}) = O(\log n)$

HEAPSORT

1. If we use a heap as a PQ and sort using it, the run-time of said algorithm is

$$\begin{aligned} T(n) &\in O(\text{initialization} + n \cdot \text{insert} + n \cdot \text{deleteMax}) \\ &= O(\Theta(1) + n \cdot \Theta(\log n) + n \cdot \Theta(\log n)) \\ &= O(n \log n). \end{aligned}$$

2. Pseudocode:

```
HeapSort(A,n)
    // heapify
    n ← A.size()
    for i ← parent(last()) down to 0 do
        fix-down(A,i,n)
    // repeatedly find maximum
    while n>1
        // 'delete' maximum by moving to end and decreasing n
        swap item at A[root()] and A[last()]
        n--
        fix-down(A, root(), n)
```

* Heapify:

Given: all items that should be on the heap

It builds the heap all at once.

How? → by fixing-down incrementally.

Heapify has run-time $\Theta(n)$.

Why? → analyze a recursive version:

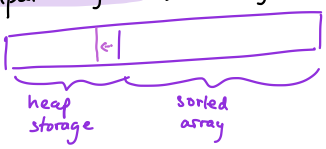
```
heapify(node i)
    if i has left child
        heapify(left child i)
    if i has right child
        heapify(right child i)
    fix-down(i)
```

Now, let
$T(n) :=$ runtime of heapify on n items.
(Assume n divisible as needed.)
Then:
size(left child) = size(right child) $= \frac{n-1}{2}$.

$$\therefore T(n) = \begin{cases} \Theta(1), & n \le 1 \\ T(\frac{n-1}{2}) + T(\frac{n-1}{2}) + \Theta(\log n), & \text{otherwise} \end{cases}$$

$$\Rightarrow T(n) \le 2T(\tfrac{n}{2}) + \Theta(\log n) \in O(n). \quad \square$$

3. Note this uses $O(1)$ auxiliary space, since we use the same input-array A for storing the heap.



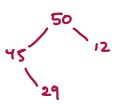
OTHER PQ OPERATIONS

We can also support the following operations:

1. "findMax" — finds the max element without removing it;
   - in a bin heap this takes $\Theta(1)$
   - since it is just the root node
2. "decreaseKey" — takes in a ref i to the location of one item of the heap and a key k_{new}, and decreases the key of i to k_{new} if $k_{new} < \text{key}(i)$
   - does nothing if $k_{new} \ge \text{key}(i)$
   - easy to do in a bin heap; just need to change the key & then call fix-down on i to its children
3. "increaseKey" — "opposite" of decreaseKey.
   - easy in bin heap, but just call fix-up instead of fix-down
4. "delete" — delete the item i (which we have a ref to).
   - for bin heap: increase value of key at i to ∞ (or findMax().key()+1);
   - then call deleteMax
   - takes $O(\log n)$

MELDABLE HEAP

A "meldable heap" is the same as a "binary heap", except it drops the structural property.



* so just that each root is the maximum of all its children

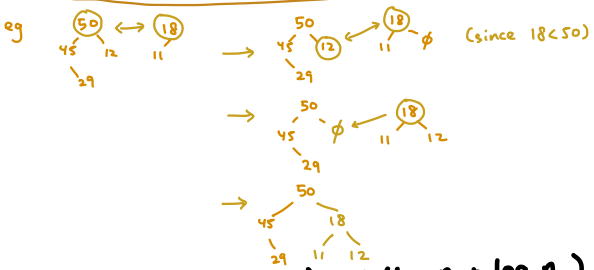
Operations for meldable heaps:

1. insert
   - create 1-node meldable heap P' w/ new k-v pair we want to add
   - call merge(P, P') w/ existing & newly-created meldable heap
2. deleteMax
   - max is root, so just remove that
   - then return merge(P_L, P_R), where P_L & P_R are the "subheaps" of the original heap

3. merge

```
MeldableHeap::merge(r1, r2)
input: r1, r2 (roots of two meldable heaps)
    - r1 ≠ NIL
output: returns root of merged heap
    if r2 is NIL then return r1
    if r1.key < r2.key then swap r1, r2
    randomly pick one child c of r1
    replace c by result of merge(r2, c)
    return r1
```

eg



AVG RUN-TIME (MERGE) = $O(\log n_1 + \log n_2)$ (L2.3)

Note that

avg run-time (merge) = $O(\log n_1 + \log n_2)$

where n_1, n_2 are the sizes of the heaps to be merged.

BINOMIAL HEAPS

FLAGGED TREES

A "flagged tree" is one where every level is full, but the root node only has a left child.

Note that a flagged tree of height h has 2^h nodes.

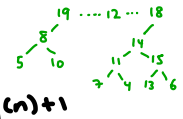
BINOMIAL HEAP (CO2.3)

A "binomial heap" is a list L of binary trees such that

1. any tree in L is a flagged tree (structural property); &
2. for any node v, all keys in the left subtree of v are no bigger than $v.key$ (order property).

PROPER BINOMIAL HEAP

We say a BH is "proper" if no two flagged trees in L have the same height.



A PBH OF SIZE n CONTAINS $\leq \log(n)+1$ FLAGGED TREES (O2.2)

Let a PBH have size n.

Then it contains at most $\log(n)+1$ flagged trees.

Proof. Let T be the tree w/ max height, say h, in the list L of flagged trees.
Then T has 2^h nodes, so $2^h \leq n$, ie $h \leq \log(n)$.
Since the trees in L have distinct heights, we have at most one tree for each height $0,\dots,h$, and so at most $h+1 \leq \log n+1$ trees. ∎

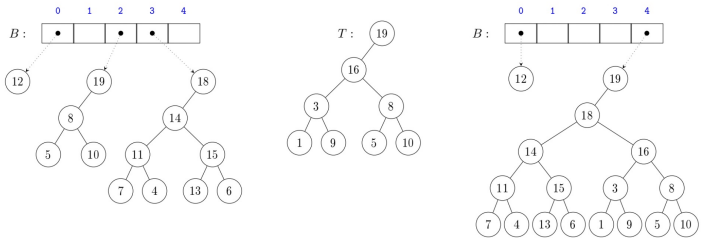
MAKING A BH PROPER

To make a BH proper, we do the following:
if the BH has two flagged trees T, T' of the same height h, then they both have 2^h nodes.

We want to combine them into one flagged tree of height $h+1$.

To do so, we use the following algorithm: (A2.11)

```
binomialHeap::makeProper()
    n ← size of the binomial heap
    for (ℓ=0; n>1; n ← ⌊n/2⌋) do ℓ++      // compute ⌊log n⌋
    B ← array of size ℓ+1, initialized at NIL
    L ← list of flagged trees
    while L is non-empty do
        T ← L.pop(), h ← T.height
        while T' ← B[h] is not NIL do
            if T.root.key < T'.root.key then swap
            T & T'
            T'.right ← T.left, T.left ← T', T.height ← h+1   // merge T with
            B[h] ← NIL, h++
        B[h] ← T
    for (h=0; h≤ℓ; h++) do                 // copy B back to the list
        if B[h] ≠ NIL then L.append(B[h])
```

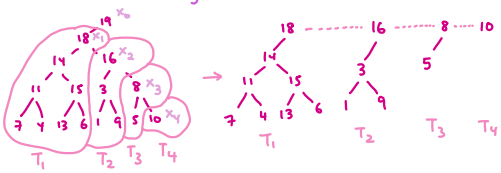


PQ OPERATIONS FOR PBHS

Each of the PQ operations can be performed in $O(\log n)$ time with makeProper.

1. merge(P_1, P_2)
   - concat lists of P_1, P_2 into one
   - then call makeProper
   - takes time $O(\log n_1 + \log n_2) \leq O(\log n)$
2. insert(k, v)
   - like w/ meldable heaps: create a single-node binomial heap, then call merge
   - takes time $O(\log n)$
3. findMax()
   - scan through L and compare keys of the roots
   - largest is just maximum
   - takes time $O(|L|) \leq O(\log n)$
4. deleteMax()
   - assume we found the max at root x_0 of flagged tree T, say w/ height h (this takes $O(\log n)$ time if heights not already known)
   - remove T from L and split it as follows:
   - let x_1 be the left child of x_0, and for $1 \leq i \leq h$, let x_{i+1} be the right child of x_i.

   - let T_i be the tree consisting of x_i & its left subtree; this is a flagged tree.
   - create a second BH P' consisting of $T_1, \dots, T_h$, and note this covers all keys in T except x_0.
   - then, merge P' into what remains of the original binomial heap P.
   - as P was proper, and the list of P' has length $h \leq \log n$, merging (and thus deleteMax) has run-time $O(\log n)$.



Chapter 3: Sorting

THE SELECTION PROBLEM

The "selection problem" is:

> "Given an array A[0,...,n-1] and an index $0 \le k < n$, select(A,k) should return the element in A that would be at index k if we sorted A".

eg if A[0,...,9] = 30 60 10 0 50 80 90 10 40 70

then the sorted array would be

0 10 10 30 40 50 60 70 80 90

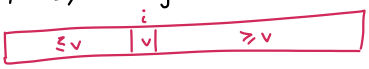
so

select(3) = 30.

PARTITION [THE FUNCTION]

The "partition" function does the following:

given the pivot-value v = A[p] and an array A[0,..,n-1], rearrange A s.t.

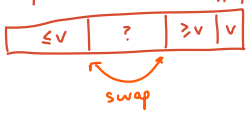


and return the pivot-index i.

Partition algorithm ("efficient in-place partition — Hoare"):

```
partition(A,p)
 // A: array of size n
 // p: integer s.t. 0≤p≤n
1.  swap(A[n-1], A[p])
2.  i←-1, j←n-1, v←A[n-1]
3.  loop
4.     do i←i+1 while A[i]<v
5.     do j←j-1 while j≥1 & A[j]>v
6.     if i≥j then break (goto 9)
7.     else swap(A[i], A[j])
8.  end loop
9.  swap(A[n-1], A[i])
10. return i
```

* we keep swapping the outer-most wrongly-positioned pairs



QUICK-SELECT

The "quick-select" algorithm:

```
quick-select1(A,k)
 // A: array of size n
 // k: integer s.t. 0≤k<n
1.  p ← choose-pivot1(A)      // for now, p=n-1
2.  i ← partition(A,p)
3.  if i=k then
4.      return A[i]
5.  else if i>k then
6.      return quick-select1(A[0,...,i-1], k]
7.  else if i<k then
8.      return quick-select1(A[i+1, i+2,..., n-1], k-i-1)
```

* intuition:

Have:

Want:

Run-time analysis of quick-select:

We analyze the # of key-comparisons.

↳ so we don't mess with constants.

In particular, partition uses n key comparisons.

Then, the run-time on an instance I is

$$T(I) = \underbrace{n}_{\text{partition}} + \underbrace{T(I')}_{\text{sub array}}$$

How big is I'?

best case: don't recurse at all → $\Theta(n)$

worst case: $|I'| = n-1$

$$\therefore T^{worst}(n) = \max_I T(I)$$
$$= n + T^{worst}(n-1)$$
$$= n + (n-1) + T^{worst}(n-2)$$
$$= \dots$$
$$= \frac{n(n+1)}{2} \in \Theta(n^2).$$

Average-case:

$$T^{avg}(n) = \frac{1}{|\mathcal{I}_n|} \sum_{I \in \mathcal{I}_n} T(I)$$
$$= \frac{1}{n} \sum_{k=0}^{n-1} \frac{1}{n!} \sum_{\pi} T(\pi, k).$$

We will do

① Creating a random quick select;

② Analyze exp. runtime; then

③ Argue that this implies bound on avg-case of quick-select.

RANDOMIZED QUICK-SELECT

1. Consider the "randomized quick-select" algorithm:

```
quick-select1 (A, k)
// A : array of size n
// k : integer s.t. 0<k<n
1.   p ← random(n)   // key step
2.   i ← partition (A, p)
3.   if i=k then
4.       return A[i]
5.   else if i>k then
6.       return quick-select1 (A[0,...,i-1], k]
7.   else if i<k then
8.       return quick-select1 (A[i+1, i+2, ..., n-1], k-i-1)
```

2. Then, what is P(pivot-index = i)?

   ⇒ pivot-value is equally likely to be any of A[0], ..., A[n-1]

   ⇒ ∴ pivot-index is equally likely to be any of 0, ..., n-1

   ⇒ $P(\text{pivot index} = i) = \frac{1}{n}$

3. We claim $T^{exp}(n) \in O(n)$.

Proof. Recall that

$$T^{exp}(n) = \max_{I} \sum_{\substack{\text{random} \\ \text{outcomes } R}} P(R) \cdot T(I,R)$$

In particular, note that

$$T(I,R) = \begin{cases} n + T(A[0,\dots,i-1], k, R') & i>k \\ n + T(\text{right subarray}, k-1-i, R'), & i<k \\ n & i=k \end{cases}$$

$\uparrow$ $\nwarrow$
$\langle A,k\rangle$ $\langle i,R'\rangle$

(we count # of comparisons)

Then,

$$\sum_R P(R)\,T(I,R) = \sum_R P(R)\,T(A,k,R)$$

$$= \sum_{(i,R')} \underbrace{P(i)}_{\frac{1}{n}} P(R') \underbrace{T(A,v,R)}_{n + \{\dots\}}$$

$$= n + \frac{1}{n}\sum_{i=0}^{n-1} \begin{cases} \sum_{R'} P(R')\,T(A_\ell, k, R') & i>k \\ \sum_{R'} P(R')\,T(A_r, k-i-1, R') & i<k \\ 0 & i=k \end{cases}$$

$$\le n + \frac{1}{n}\sum_{i=0}^{n-1} \begin{cases} \max_{k'} \max_{A_\ell} \sum_{R'} P(R')\,T(A_\ell, k', R') & i>k \\ \max_{k'} \max_{A_r} \sum_{R'} P(R')\,T(A_r, k', R') & i<k \\ 0 & i=k \end{cases}$$

$$= n + \frac{1}{n}\sum_{i=0}^{n-1} \begin{cases} T^{exp}(i) & i>k \\ T^{exp}(n-i-1) & i<k \\ 0 & i=k \end{cases}$$

$$\le n + \frac{1}{n}\sum_{i=0}^{n-1} \max\{T^{exp}(i),\ T^{exp}(n-i-1)\}$$

Then, we show $T^{exp}(n) \le 8n$.

Proof. By induction.

$n=1$: Comp = $0 < 8(1) = 8$.

step:

$$T(n) \le n + \frac{1}{n}\sum_{i=0}^{n-1} \max\{8i,\ 8(n-i-1)\}$$

bad	good	bad
$n/4$	$3n/4$	

$$\le n + \frac{1}{n}\sum_{i\ \text{good}} 8\left(\tfrac{3}{4}n\right) + \frac{1}{n}\sum_{i\ \text{bad}} 8n$$

$$= n + \frac{1}{n}\cdot\frac{n}{2}(6n) + \frac{1}{n}\cdot\frac{n}{2}(8n)$$

$$= n + 3n + 4n = 8n. \quad \#$$

∴ $T^{exp}(n) \in O(n)$. ∎

QUICK-SORT

Pseudocode:

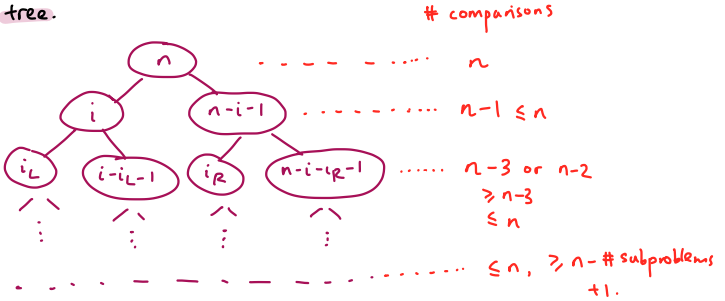
```
quick-sort1(A)
// A: array of size n
1. if n≤1 then return
2. p ← choose-pivot1(A)
3. i ← partition(A,p)
4. quick-sort1(A[0,...,i-1])
5. quick-sort1(A[i+1,...,n-1])
```

There are better implementations of this idea.

Runtime:

$T(n) = n + T(i) + T(n-i-1)$.

But there's a simpler method: the recursion tree.



Thus

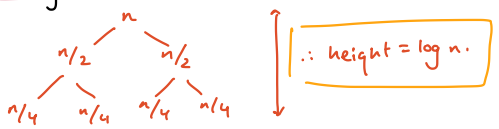
comparisons ≤ n · # layers = n · height of the recursion tree.

So what can we say about the recursion tree's height?

① Worst case: height = $\Theta(n)$
- tight if array is sorted.
- we need n recursions.

Thus run-time $\in \Theta(n^2)$.

② Best case: if the pivot-index is $\sim \frac{n}{2}$ always.



Thus run-time $\in \Theta(n \log n)$.

We can improve QuickSort's run-time by

① Not passing sub-arrays, but instead passing "boundaries";

② Stopping recursion early;
- stop recursing when array of subproblem is ≤ 10
- then use insertion-sort to sort the array
- which has $\Theta(n)$ best-case run time if the array is (almost) sorted

③ Avoid recursions;

④ Reduce auxiliary space;
- in the worst case (currently), the auxiliary space is $|S| \in O(n)$.
- but if we put the bigger subproblem on the stack, the space can be reduced to $|S| \in O(\log n)$.

⑤ Choose the pivot-index efficiently;
- don't let $p = n-1$. (run-time $= O(n^2)$)
- use "median-of-3": use the median of $\{A[0], A[\lfloor \frac{n}{2} \rfloor], A[n-1]\}$ as the pivot-value

AVERAGE-CASE QUICK-SORT

We accomplish this via randomization of the algorithm:

```
RandQSort(A)
// A: array of size n
1. if n≤1 then return
2. p ← random(n)
3. i ← partition(A,p)
4. quick-sort1(A[0,...,i-1])
5. quick-sort1(A[i+1,...,n-1])
```

Then,

$T^{exp}(n)$ = exp # of comparisons of the algo.

Note that

$$T(A,R) = n + T(\underbrace{\text{left sub array}}_{\text{size } i}, R') + T(\underbrace{\text{right sub array}}_{\text{size } n-i-1}, R').$$

($T(A,R)$: instance; R: outcomes)

Hence

$$\sum_R P(R)T(A,R) = n + \frac{1}{n}\sum_{i=0}^{n-1} T^{exp}(i) + \frac{1}{n}\sum_{i=0}^{n-1} T^{exp}(n-i-1)$$
$$= n + \frac{2}{n}\sum_{i=2}^{n-1} T^{exp}(i),$$

and so

$$T(n) = \begin{cases} 0, & n \leq 1 \\ n + \frac{2}{n}\sum_{i=2}^{n-1} T(i), & \text{otherwise.} \end{cases}$$

We claim $T(n) \in O(\log(n))$, so the expected run-time of RandQSort is in $O(n \log n)$, and so the average-case run time of QSort is in $O(n \log n)$.

Proof. Specifically, we prove $T(n) \leq 2 \cdot n \cdot \ln(n)$.

By induction. Base ($n=1$) is trivial.

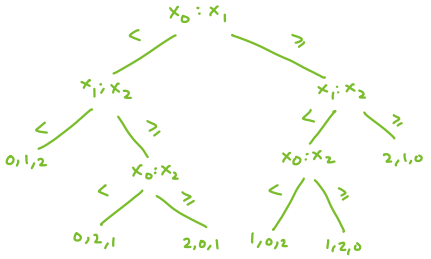
Step:
$$T(n) \leq n + \frac{2}{n}\sum_{i=2}^{n-1} 2i\ln(i)$$
$$= n + \frac{4}{n}\sum_{i=2}^{n-1} i\ln(i)$$
$$\leq n + \frac{4}{n}\int_2^n x\ln x\, dx \quad \text{(since } x\ln x \text{ is increasing)}$$
$$\leq n + \frac{4}{n}\left(\frac{1}{2}n^2\ln(n) - \frac{1}{4}n^2\right)$$
$$= n + 2n\ln(n) - n$$
$$= 2n\ln(n). \quad \blacksquare$$

ANY COMPARISON BASED SORTING ALGORITHM USES $\Omega(n \log n)$ KEY-COMPARISONS IN THE WORST-CASE

As above.

Proof. Fix an arbitrary comparison-based sorting algorithm $\mathcal{A}$, and consider how it sorts an instance of size n.

Since $\mathcal{A}$ uses only key-comparisons, we can express it as a decision tree Υ.



A decision tree for an algo to sort 3 elements x_0, x_1, x_2 with ≤ 3 key-comparisons.

For each sorting perm π, let I_π be an instance that has distinct items and sorting perm π (ie $I_\pi = \pi^{-1}$).
Executing $\mathcal{A}$ on I_π leads to a leaf that stores π.
Note that no two sorting perms can lead to the same leaf, as otherwise the output would be incorrect for one (as the items are distinct).

We then have $n!$ sorting perms, and hence at least $n!$ leaves that are reached for some I_π.

Let h be the largest layer-number of a leaf reached by some I_π.

Since Υ is binary, for any $0 \leq \ell \leq h$ there are at most 2^ℓ leaves in layers $0, \dots, \ell$.

Therefore $2^h \geq n!$, or

$$\begin{aligned} h \geq \log(n!) &= \log(n(n-1)\cdots 1) \\ &= \log(n) + \log(n-1) + \cdots + \log(1) \\ &\geq \log(n) + \log(n-1) + \cdots + \log(\lceil \tfrac{n}{2} \rceil) \\ &\geq \log(\tfrac{n}{2}) + \log(\tfrac{n}{2}) + \cdots + \log(\tfrac{n}{2}) \\ &\geq \tfrac{n}{2} \log(\tfrac{n}{2}) \\ &= \tfrac{n}{2}\log(n) - \tfrac{n}{2} \in \Omega(n \log n). \end{aligned}$$

Finally, consider the sorting perm π that has its leaf on layer h.
Executing algorithm A on I_π hence takes $h \in \Omega(n \log n)$ key-comparisons, so the worst-case bound holds. ☒

SORTING INTEGERS

BUCKET-SORT

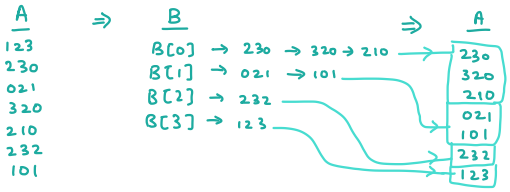
Bucket-sort can be used to sort a collection of integers by a specific "position" of digit.

eg the last digit 12345

Pseudocode:

```
bucket-sort (A, n←A.size, ℓ←0, r←n-1, d)
// A: array of size ≥n with numbers with
      m digits in {0,..,R-1}
// d: 1≤d≤m
// output: A[ℓ...r] is sorted by "d" digit.
1.  initialize an array B[0...R-1] of empty
    lists
2.  for i←ℓ to r do
3.      append A[i] at the end of          // move array-items
        B [dth digit of A[i]]                 to buckets
4.  i←ℓ
5.  for j←0 to R-1 do                      // move bucket-items
6.      while B[j] is not empty do            to array
7.          move first element of B[j]
            to A[i]
8.          i++
```

eg



See that

① Run-time = $\Theta(n+R)$; and

② Auxiliary space = $\Theta(n+R)$.

MSD-RADIX-SORT

"MSD (Most Significant Digit) radix sort" can be used to sort multi-digit numbers.

Pseudocode:

```
MSD-radix-sort (A, n←A.size, L←0, r←n-1, d←1)
// A: array of size ≥n, contains numbers with
      m digits in {0,...,R-1}, m, R are global
      variables
// ℓ, r: range (ie A[ℓ...r]) we wish to sort
// d: digit we wish to sort by
1.  if ℓ<r then
2.      bucket-sort(A, n, ℓ, r, d)
3.      if d<m then
            // find sub-arrays that have the same
               dth digit and recurse
4.          int ℓ'←ℓ
5.          while ℓ'<r do
6.              int r'←ℓ'
7.              while r'<r & dth digit of A[r'+1] = dth digit of
                A[ℓ'] do r'++
8.              MSD-radix-sort(A,n,ℓ', r', d+1)
9.              ℓ'←r'+1
```

Note that

① run-time = $\Theta(mRn)$; &

② auxiliary space = $\Theta(n+R+m)$.

LSD-RADIX-SORT

"LSD-Radix-Sort" is a better way of sorting multi-digit numbers (than MSD) because

① it has a faster runtime;

② it uses less auxiliary space; and

③ it uses no recursion.

Pseudocode:

```
LSD-radix-sort (A, n←A.size)
// A: array of size n, contains m-digit
      radix-R numbers, m, R are global
1.  for d←m down to 1 do
2.      bucket-sort(A,d)
```

Clearly

① run-time = $\Theta(m(n+R))$; &

② auxiliary space = $\Theta(n+R)$.