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 $\neq 0$

$\Rightarrow f(n) \notin o(g(n))$. ∎

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

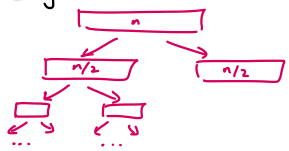
Suppose $f(n) \in \omega(g(n))$. Then $f(n) \in \Omega(g(n))$.

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

ADT PRIORITY QUEUES

A "priority queue" stores items that have a priority, or key.

Operations:

1. insert (a given item & priority as a k-v pair)
2. deleteMax (return item with largest priority)
3. size, isEmpty

We can use a PQ to sort:

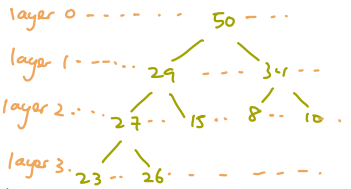
```
PQ-Sort(A[0, ..., n-1])
    init PQ to an empty PQ
    for i←0 to n-1 do
        PQ.insert(A[i])
    for i←n-1 down to 0 do
        A[i] ← PQ.deleteMax()
```

The run-time of the above algorithm is $O(\text{initialization} + n \cdot \text{insert} + n \cdot \text{deleteMax})$.

BINARY HEAPS

"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"

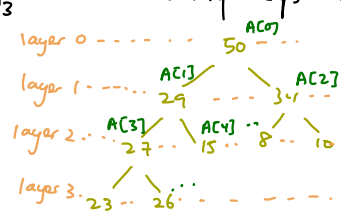


Each node is "shorthand" for a reference to a <key-value> pair:

"50" ⟷ priority = 50, data = <...>

Note that binary heaps have height $O(\log n)$.

We store binary heaps using an array:



A

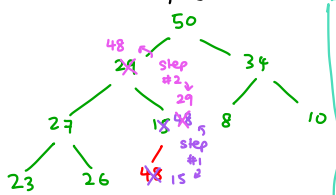
0	1	2	3	4	5	...
50	29	34	27	15	8	...

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

INSERT IN HEAPS

To insert into a heap, we just place the new key at the first free leaf.

We also employ "fix-up":

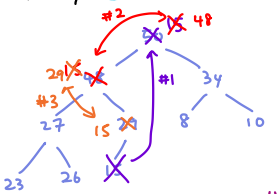


Pseudocode

```
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:

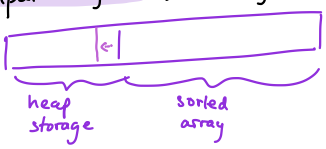
```
heapify(node i)
    if i has left child
        heapify(left child i)
    if i has right child
        heapify(right child i)
    fix-down(i)
```

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:

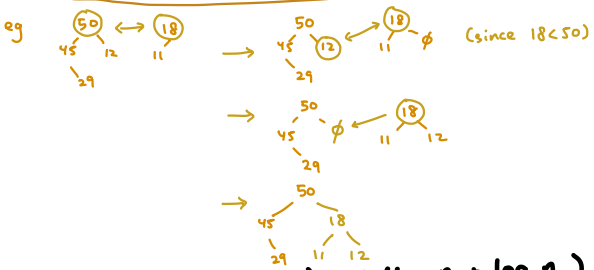
① insert
- create 1-node meldable heap P' w/ new k-v pair we want to add
- call merge(P, P') w/ existing & newly-created meldable heap

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 (CD2.3)

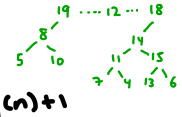
A "binomial heap" is a list L of binary trees such that

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 ≤ log(n)+1 FLAGGED TREES (CD2.2)

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 $2^h \le n$,
ie $h \in \log(n)$.
Since the trees in L have distinct heights, we have at most one tree for each height $0, \dots, h$, and so at most $h+1 \le \log n + 1$ trees. ∎

MAKING A BH PROPER

To make a BH proper, we do the following:

① if the BH has two flagged trees T, T' of the same height h, then they both have 2^h nodes.

② We want to combine them into one flagged tree of height h+1.

③ To do so, we use the following algorithm: (CA2.11)

```
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
      if B[h] ≠ NIL then L.append(B[h])   // copy B back to the list
```