Ch.2 Algorithm Analysis

2.1 Time Complexity Analysis

Algorithm's run time as a function of input size

$T(n)$ worst case

we care about the "asymptotic behavior"

- how $T(n)$ grows when n is large
- only care about the "order" instead of exact function

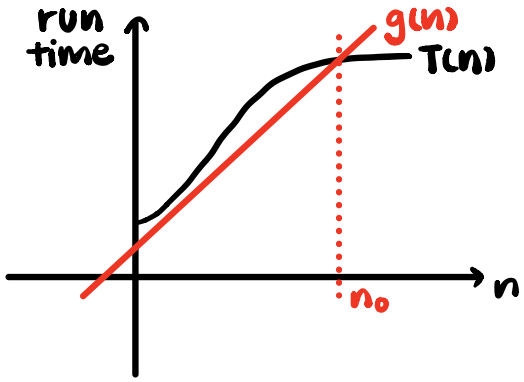
Def. $T(n)$ is $O(g(n))$ if $\exists$ constant $c>0$, $n_0 \geq 0$ s.t. $T(n) \leq c \cdot g(n)$ $\forall$ $n \geq n_0$

Big-O

Ex. $T(n) = 100n + 500$

$T(n) = O(n)$

$T(n) = O(n^2)$



Upper bound of time complexity

Asymptotic lower bound:

Big Omega

$T(n) = \Omega(f(n))$

if there exists $c > 0$, $n_0 \geq 0$ s.t.

$T(n) \geq c \cdot f(n) \quad \forall \; n \geq n_0$

$T(n) = \Omega(n)$

$T(n) = \Omega(\sqrt{n})$

$T(n) \neq \Omega(n^2)$

Asymptotic tight bound

Def. $T(n) = \Theta(f(n))$ if

$T(n) = \Omega(f(n))$ and $T(n) = O(f(n))$

Ex. $T(n) = 100n$

$T(n) = \Theta(n)$

$\neq \Theta(\sqrt{n})$

$\neq \Theta(n^2)$

$T(n) = \Theta(T(n))$

Ex. $T(n) = Rn^2 + qn + r$ $\quad p, q, r$ const.

$T(n) = \Theta(n^2)$ $\quad T(n) = O(n^2)$ $\quad T(n) = \Omega(n^2)$

choose c large enough $(c > p+q+r)$

$T(n) \le c \cdot n^2$ when $n \ge 2$

$T(n) = O(n^3)$, $T(n) = O(n^4)$, $T(n) \neq O(n)$

$T(n) = \Omega(n^2)$, $T(n) = \Omega(n)$, $T(n) \neq \Omega(n^3)$

Properties :

- $f = O(g) \Rightarrow c \cdot f = O(g)$
- $f = O(h)$, $g = O(h) \Rightarrow f + g = O(h)$

Pf. $f = O(h) \Rightarrow$ there exist c_1, n_1 such that

$f(n) \le c_1 \cdot h(n) \quad \forall\ n \ge n_1$

$g = O(h) \Rightarrow$ there exist c_2, n_2 such that

$g(n) \le c_2 \cdot h(n) \quad \forall\ n \ge n_2$

Take $c = c_1 + c_2$, $n' = \max(n_1, n_2)$.

Then $f(n) + g(n) \le (c_1 + c_2) \cdot h(n)$

$= c \cdot h(n) \quad \forall\ n \ge n'$

$\Rightarrow f + g = O(h)$.

- $f = O(h_1)$, $g = O(h_2) \Rightarrow f + g = O(\max\{h_1, h_2\})$

Polynomial $T(n) = a_d n^d + a_{d-1} n^{d-1} + \dots + a_1 n + a_0$

$T(n) = \Theta(n^d) \quad (a_d \neq 0)$

$\log_a n = \Theta(\log_b n)$ $\quad a \neq b$, a, b const

Pf. $\log_a n = \dfrac{\log_b n}{\log_b a}$ ↑ const.

$\log_a n = O(n^\varepsilon) \quad \forall\ \varepsilon > 0$

e.g. n $\log_2 n$ $n^{0.1}$

Thm.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & f(n) = O(g(n)) \\ c & (0 < c < \infty) \quad f(n) = \Theta(g(n)) \\ \infty & f(n) = \Omega(g(n)), \quad f(n) \neq \Theta(g(n)) \end{cases}$$

$$\Updownarrow$$

$$g(n) = O(f(n))$$

Time Complexity of G-S algorithm

Array : + access A[i] in constant time O(1)
- can't dynamically +/- an element

Linked list : + dynamically +/- an element
- access A[i] in O(n) time

G-S algorithm

input : ManPref[m, i], WomanPref[m, i] (2D array)

while (∃ unmatched m) ①

w ← next woman in m's preference list to whom m hasn't proposed ②

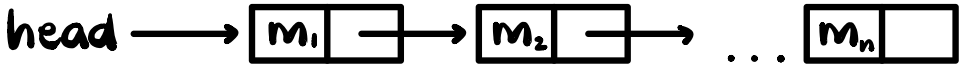
if (w unmatched) ③

match (m, w)

else if (w prefers m to her current partner m')

remove (m', w), match (m, w)

① store unmatched men in linked list



② use next(m) : next position m will propose

initially next(m) = 1 ∀ m

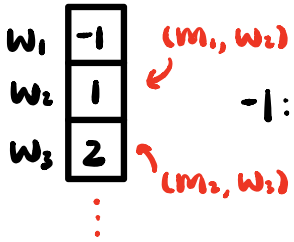
w ← ManPref[m, next(m)]

m_1 : | w_3 | w_1 | w_2 |

③ store current matching in array

match(w) = current partner of w

n elements



-1: unmatched

④ w, m, m'

how to check whether w prefers m to m'

(m_1 m_2)

find m_2 m_1 in O(n)

⇒ inverse pref. list

$B[w, m] \overset{?}{>} B[w, m']$

↓

O(1)

each iteration of G-S O(1) time

→ worst case: $O(n^2)$ iterations

total time complexity $\underbrace{O(n^2)}_{O(N)}$

input size $N = O(n^2)$ n : # m/w

linked list

Big-O

- Finding an element T (int) in an array A.

```
for i = 0, 1, ... n-1
    if A[i]
        return i
return false
```

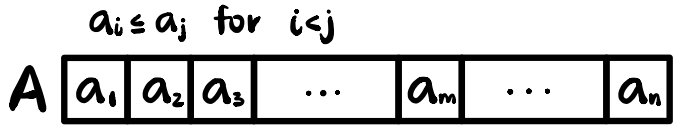
O(n)

$\Theta(n)$ time

- What if A is sorted?

  find an element in O(logn) time

  binary search:



compare a_m with T

if $a_m = T$ return a_m

if $a_m < T$ T is in $[a_{m+1}, \ldots, a_n]$

if $a_m > T$ T is in $[a_1, \ldots, a_{m-1}]$

maintain a "search image" $[\ell, r]$

Binary search:

initial $\ell = 0$, $r = n$

while $(\ell < r)$

$mid = \lfloor \frac{\ell + r}{2} \rfloor$

if $T < a_{mid}$ $r = mid - 1$

if $T > a_{mid}$ $\ell = mid + 1$

if $T = a_{mid}$ return mid

return -1 (T not found)

range $[l, r]$

initial $\underset{(n)}{l-r+1} \xrightarrow{/2} \leq n/2 \xrightarrow{/2} \leq n/4 \cdots$

after k iters, range $\rightarrow \frac{n}{2^k}$

when $k = \log_2 n$, range $= 1$

so Bsearch = $O(\log n)$ time

Target sum problem

- 2 sum: given array A, find 2 elements that sum to T

- naïve alg: try all pairs $(i, j) \Rightarrow O(n^2)$

- assume A is sorted:

  Alg. 1: for each $A[i]$, find whether $T - A[i]$ is in A (excluding $A[i]$)

  $\rightarrow O(n \log n)$ "search image"

  Alg. 2: maintain a range of valid elements $[l, r]$

  (if $a_i + a_j = T$, then $a_i, a_j \in [l, r]$)

2-sum for sorted array

initial $l=0$, $r=n$

while $(l<r)$

mid $= \lfloor \frac{l+r}{2} \rfloor$

if $a[l]+a[r]>T$ $\quad r=r-1$

if $a[l]+a[r]<T$ $\quad l=l+1$

} range reduced by 1 each iter → O(n)

if $a[l]+a[r]=T$ $\quad$ return l, r

return -1

2-sum w/o sorted array

step 1: sort array O(nlogn)

step 2: run above alg. O(n)

} O(nlogn) alg.

2-sum w/ hash table

push $a_1 - a_n$ to hash table

for each a_i, find whether $a_i - T[i]$ is in hash table

O(1) → O(logn)

2.4 Stack / Queue / Priority Queue

Stack: last-in, first-out

- S.push(a): push a into S
- S.pop(): return the "latest pushed" element, remove this element

Queue: first-in, first-out

- Q.push(a): push a into Q
- Q.pop(): return the "first pushed" element, remove this element

Priority Queue:

- Q.push(a): push a into Q
- Q.pop(): return the element w/ smallest (largest) value, remove this element

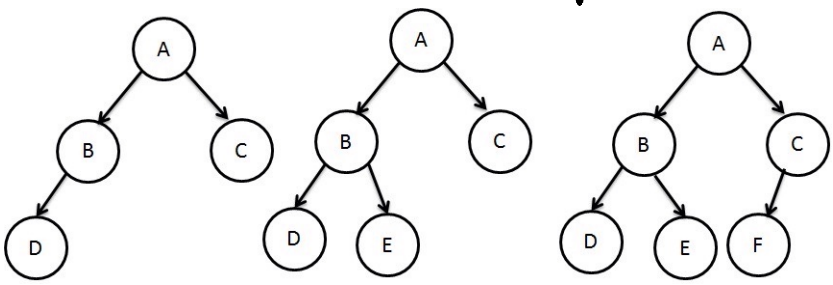
Implement priority queue:

- naïve way 1:
  - push(a): add a to the array O(1)
  - pop(): find element w/ smallest value O(n)
- naïve way 2:
  - push(a): add a to the array, sort array O(n)
  - pop(): return the first element O(1)

Heap to implement priority queue:

O(logn) to push and pop

Heap: store elements in a "complete binary tree"



for each index i:

node i's parent = $L i/2 \rfloor$

node i's left child = $i \cdot 2$

node i's right child = $i \cdot 2+1$

maintain "min-heap" property:

each node's value ≥ its parent's value

alg: push(x) O(logn)

```
n = n+1
A[n] = x
i = n
while (i ≠ 1 and A[⌊i/2⌋] > A[i])
    exchange A[⌊i/2⌋], A[i]
    i = ⌊i/2⌋
```

pop(): find min and remove O(logn)