Solving recurrence: Substitution: guess upper bound, solve with induction.

Algorithms midterm 1

بسم الله الرحمن الرحيم

Maha Alkhairy

Topics: recurrence, Divide and Conquer, Greedy, Dynamic Programming

* Asymptotic notation *

time complexity:

- $T(n) = \Omega(f(n))$ if $\exists$ constants $c>0$ and $n_0 > 0$ such that $T(n) \geq c f(n)$ $\forall$ $n \geq n_0$
- $T(n) = O(f(n))$ if $\exists$ constants $c>0$ and $n_0>0$ such that $T(n) \leq c.f(n)$ $\forall n \geq n_0$
- $T(n) = \Theta(f(n))$ if $\exists$ constants $c>0$ and $n_0>0$ such that $T(n) = c.f(n) \forall n \geq n_0$

$O(1) < O(\log\log n) < O(\log n) < O((\log n)^c) <$ ($c>1$)

$O(n^c) < O(n) < O(n \log n) = O(\log_n n^n) = <$ ($0<c<1$)

$O(\log n!) < O(n^2) < O(n^c) < O(c^n) <$ ($c>1$, $c>1$)

$O(n!)$

Master Theorem:

$T(n) = aT(\lceil n/b \rceil) + O(n^d)$

$a>0, b>1, d>0$; a = # subprobs

n/b = size of subproblems ;

n^d = time for combination

$$T(n) = \begin{cases} O(n^d) ; & d > \log_b a \\ O(n^d \log n) ; & d = \log_b a \\ O(n^{\log_b a}) ; & d < \log_b a \end{cases}$$

mergesort

proof of correctness

use induction for proof of correctness of any divide and conquer algorithm

- base case: $n=1$: algo does nothing → correctly sorts

Induction: assume alg sort arrays size $\boxed{n-1}$ correctly there $A[1,\ldots m]$ $a[m+1,\ldots n]$ are correctly sorted since size $\leq n-1$

lemma: Merge correctly merges 2 sorted arrays

run time: merge sort

$T(n) \leq 2T(n/2) + c.n$

$O(n \log n)$

Divide & Conquer

general recipe:

1. Divide: break into subproblems that are instance of same problem type
2. Conquer: solve subproblems recursively. if small enough; solve straight forward.
3. Combine: combine solutions cleverly!

Example:

```
def multiply(x, y):
    input: n-bit pos x, y ints
    out: product

    if n=1 return xy
    x_L, x_R = leftmost ⌈n/2⌉, rightmost ⌊n/2⌋
               bits of x
    y_L, y_R = ...

    P1 = multiply(x_L, y_L)
    P2 = multiply(x_R, y_R)
    P3 = multiply(x_L + x_R, y_L + y_R)
    return P1 * 2^(2⌊n/2⌋) + (P3 - P1 - P2) * 2^⌊n/2⌋ +
           P2
```

mergeSort: $(n \log n)$

```
def mergeSort(a[1,...n]):
    if n>1:
        return merge(mergesort(a[1...n/2]),
                     mergeSort(a[n/2+1, ... n]))
    else:
        return a
```

∘ : means concatenate

```
def merge(x[1,...k], y[1,...l]):
    if k=0: return y[1,...l]
    if l=0: return x[1,...k]
    if x[1] ≤ y[1]:
        return x[1]∘merge(x[2,...k],
                          y[1,...l])
    else
        return y[1]∘merge(x[1,...k],
                          y[2,...l])
```

Greedy algorithm: recipe

make whichever move seems best at moment and not worry too much about future consequence.

build solution piece by piece. choose next piece which offers most obvious and immediate effect.

try with very simple example first. ✗

Proof of correctness

exchange argument	greedy stays ahead
• assume optimal solution O is 'out of order' • exchange two of out of order elements to get O' • argue cost of O' is smaller than cost of O to get contradiction.	A: greedy out sol size k elem i O: optimal size m elem j $m \le k$ Show $F[i_r] \le F[j_r]$ ∀ $1 \le r \le k$ if $m > k$ add $j_{k+1} \to m$ contradiction.

cut property: cut is any partition of vertices to two groups.
in MST: cut property says safe to add lightest edge in cut

MST: min spanning tree: connected graph with no cycles

Kruskals: repeatedly choose edge lightest that doesn't produce cycle

(Union-find) keep track of disjoint sets of n elements $\{x_1, \ldots x_n\}$

1) makeset(x): make a set {x} O(1)
2) Union(A,B): O(1)
3) Find(x) return set containing x O(logn)

Kruskals: MST

```
for u∈V:
  makeset(u)
X = { }
Sort edges by weight
for edge {u,v} in E:
  if find(u) ≠ find(v):
    add edge {u,v} to X
    union(u,v)
```

Prim: (G, w) :- MST

```
for u in V:
    cost(u) = ∞
    prev(u) = nil
any node u0
cost(u0) = 0
H = make queue(V)
While H not empty:
   v = delmin(H)
   for {v,z} ∈ E:
     if cost[z] > w(v,z):
        cost(z) = w(v,z)
        prev(z) = v
        decrease key(H,z)
```

I ∋ ∅

Matroid A = (S, I) such that
1) S is finite
2) I is non-empty; hereditary
3) exchange property: A ∈ I, B ∈ I, |A| < |B|, ∃ x ∈ B − A, A ∪ {x} ∈ I

Matroid: A ∈ I is a basis if A ⊄ of a set in I (not a proper subset)

graphic matroid:
• non emptiness − ∅ independent (V, ∅) no cycles
• (V, I) acyclic so is (V, I') ∀ I' ⊆ I
• exchange: don't create cycles

Dynamic Programming: Correctness: by induction

identify collection of subproblems and tackle.

Common subproblems: (runtime).

1. $x_1 \ldots x_n$ input subproblem $x_1 \ldots x_j$ O(n)
2. $x_1 \ldots x_n$; $y_1 \ldots y_m$ input $x_1 \ldots x_i$, $y_1 \ldots y_j$ } O(mn)
3. ~~$x_1 \ldots x_n$ sub $x_i \ldots x_j$~~ O(n²)
4. tree input: subproblem choose a node in tree and treat subtree as subproblem.

trick: choose subproblems such that all vital information is remembered and carried forward.

example: optimal binary search trees. want min total access cost. (O(n³))

```
optbst(A, f):
  F[1..n][1..n] = computeF(f)
  for i ← 1 to n+1:
     OPT[i, i−1] ← 0
  for d ← 0 to n−d:
     OPT[i, i+d] ← F[i, i+d] + min_{i≤r≤i+d} {OPT[i, r−1] + OPT[r+1, i+d]}
  return OPT[1, n]

computeF(f[1..n]):
  for i in range(1→n):
     F[i, i−1] ← 0
     for j ← i to n:
        F[i, j] ← F[i, j−1] + f[j]
```

Alternating Sign Subseq:

```
Sign(a) = 1 if a > 0; 0 if a < 0
ASS(A[1..n], B[1..n])
F[i, j, k] = 0 ∀ i, j, k
for i ← 1 to n:
  for j ← 1 → n:
    if A[i] = B[j] and Sign(A[i]) = 1
       F[i, j, 1] ← max(1 + F[i−1, j−1, 0],
                        F[i−1, j, 1],
                        F[i, j−1, 1])
       F[i, j, 0] ← max(F[i−1, j, 0],
                        F[i, j−1, 0])
    elif A[i] = B[j] and Sign() = 0
       same but flip 1 with 0
    else:
       F[i, j, 1] ← max{F[i−1, j, 1], F[i, j−1, 1]}
       same for F[i, j, 0] but flip
return max{F[n, n, 0], F[n, n, 1]}
```