Cheat Sheet - Exam 2

Wednesday, November 20, 2019 1:12 PM

Divide & Conquer

3 Parts to Strategy
- Divide: Break problems into several smaller instances of same problem.
- Conquer: if smaller stage trivial, solve otherwise divide again.
- Combine: Combine results for smaller probs into solution for large prob.

Minimum Cut Property
- Min s-t cut is the s-t cut w/ min capacity.

Max-Flow, Min Cut Theorem
- In every flow network, max s-t flow is equal to the min capacity of an s-t cut.

$$v(f) = f^{out}(A) - f^{in}(A) = \sum_{e \text{ out } A} f(e) - \sum_{e \text{ into } A} f(e)$$

Ford Fulkerson - Algorithm where you choose a path, bottleneck capacity is the limiting edge.

QuickSort
- Runtime of Quicksort: Worst: $O(n^2)$, Avg: $O(n \log n)$, Best: $O(n \log n)$
- Proof:

Correctness Use the loop invariant to prove correctness of PARTITION:

initialization Before the loop starts, all the conditions of the loop invariant are satisfied, because r is the pivot and the subarrays $A[p..i]$ and $A[i+1..j-1]$ are empty.

maintainance While the loop is running, if $A[j] \leq$ pivot, then $A[j]$ and $A[i+1]$ are swapped and then i and j are incremented. If $A[j] >$ pivot, then increment only j.

termination When the loop terminates, $j = r$, so all elements in A are partitioned into one of the three cases: $A[p..i] \leq$ pivot, $A[i+1..r-1] >$ pivot, and $A[r] =$ pivot.

The last two lines of PARTITION move the pivot element from the end of the array to between the two subarrays. This is done by swapping the pivot and the first element of the second subarray, i.e., by swapping $A[i+1]$ and $A[r]$.

MergeSort
- Divide array into two every iteration

A = [5, 2, 4, 7, | 1, 3, 2, 6]

[5, 2, 4, 7] [1, 3, 2, 6]

[5, 2] [4, 7] [1, 3] [2, 6]

[5] [2] [4] [7] [1] [3] [2] [6]

- Runtime of MergeSort
  - Best: $O(n \log n)$
  - Worst: $O(n \log n)$

Median of Medians
- Used for finding a good pivot for QuickSort
- Randomly select 3 items from array, find median, use as pivot.
- Runtime: $O(n)$

Partitioning - Select last element as pivot, select pivot at random, MoM.

QuickSelect
- Runtime: Worst: $O(n^2)$, Best: $O(n)$, Avg: $O(n)$
- Algo to select k^{th} smallest number in array
- Perform Binary Search for k^{th} element

Binary Search
- A = Sorted array, if $n >$ len(A)//2, n is in the right half of the array if $n <$ len(A)//2, n is in the left half of the array

Runtime:
- Best: $O(1)$
- Worst: $O(\log n)$

Knapsack
- Bag w/ weight W, n items, maximize profit and $\sum w_i$ of items $\leq W$
- Ex. $P = \{1, 2, 5, 6\}$ $w = \{2, 3, 4, 5\}$ Table = V

Dynamic Programming
Memorization: Top-down (Iterative)

Tabulation: Bottom-Up (Iterative)

Not Useful for:
- Probs w/out overlapping sub-probs
- Useful for probs with.

LCS
- Finding the longest sub string of two strings not in a particular order
  - Ex. A = 'AGGTAB' B = 'GXTXAYB' LCS = 'GTAB'

Pseudocode on page 2

P_i	W_i	0	0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5	6	6	7	8

$$V[i,w] = \max\{V[i-1,w], V[i-1, w - w[i]] + P[i]\}$$

Partition for QuickSort

```
partition(arr,low,high):
i = ( low-1 )      # index of smaller element
pivot = arr[high]  # pivot

for j in range(low , high):

    # If current element is smaller than or
    # equal to pivot
    if   arr[j] <= pivot:

        # increment index of smaller element
        i = i+1
        arr[i],arr[j] = arr[j],arr[i]

# swap elements
arr[i+1],arr[high] = arr[high],arr[i+1]
return ( i+1 )
```

Cont. LCS Ex.

		A	G	G	T	A	B
G				1			
X							
T					2		
X							
A						3	
Y							
B							4

$\Rightarrow$

∴ The LCS = "GTAB", length: 4

```
# Dynamic Programming implementation of LCS problem

def lcs(X , Y):
    # find the length of the strings
    m = len(X)
    n = len(Y)

    # declaring the array for storing the dp values
    L = [[None]*(n+1) for i in xrange(m+1)]

    """Following steps build L[m+1][n+1] in bottom up fashion
    Note: L[i][j] contains length of LCS of X[0..i-1]
    and Y[0..j-1]"""
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0 :
                L[i][j] = 0
            elif X[i-1] == Y[j-1]:
                L[i][j] = L[i-1][j-1]+1
            else:
                L[i][j] = max(L[i-1][j] , L[i][j-1])

    # L[m][n] contains the length of LCS of X[0..n-1] & Y[0..m-1]
    return L[m][n]
#end of function lcs
```

Weighted Interval Scheduling

- Set of requests, 1...n, request i has finish time f_i, start s_i, weight w_i
- Select subset of requests where maximize weights, not # of requests.
- Sort by finish time
- P(j) = # of requests not overlapping with j where the event is $< j$.

Is request j part of optimal soln'

Yes: No requests between P(j), j can be part of optimal soln'

No: If request j is not part of soln', then soln' in requests 1, ..., j-1

Sequence Alignment

- Shows the optimal alignment
- Score of aligning two strings.
- Needleman-Wunch algorithm
- Take Min(nw, w, n) of ij in the table.

Ex. X = TALE
Y = APE

Final Alignment

T	A	L	E
d	I	s	I
–	A	P	E

Operations

Sub: Case 1- X = A, Y = A ⇒ Cost = 0

Case 2- X = A, Y = B ⇒ Cost = 1

Insert: Insert from Source to Target

Delete: Delete from Source

In/del: Cost = 1

X/y	_	A	P	E
–	0	1	2	3
T	1	1	2	3
A	2	1	2	3
L	3	2	2	3
E	4	3	3	2

Another Example (A1536)

Y = target

	_	T	H	E	R	E
_	0	1	2	3	4	5
T	1	0	1	2	3	4
H	2	1	0	1	2	3
E	3	2	1	0	1	2
I	4	3	2	1	1	2
R	5	4	3	2	1	2

X = source

THEIR_
THE_RE

What is alignment

← insert into source, don't show R, add gap

E

K sub, use R
↑ delete from source, adds gap to target
K sub, use E
K sub, use H
K sub, use T

Bipartite Matching

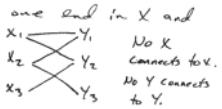
Bipartite Matching

Application of max flow. Set up network in a special way.

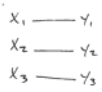
Ex: Matching candidates to jobs
Matching residents to med schools

Bipartite Graph

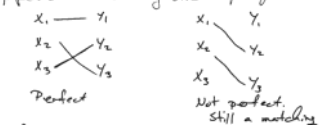
A graph G=(V,E) is bipartite if its vertex set V is partitioned into sets X and Y s.t. every edge has one end in X and one end in Y.



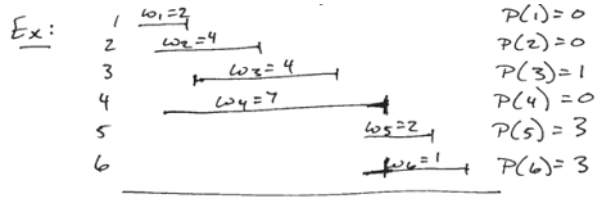
Matching - A matching in a graph G=(V,E) is a set of edges $M \subseteq E$ where every vertex appears in at most one edge of M.



M is a perfect matching if every V appears in exactly one edge of M



Ex:



P(1) = 0
P(2) = 0
P(3) = 1
P(4) = 0
P(5) = 3
P(6) = 3

Ex: In graph above, assume j=5 is part of solution. P(5) = 3
Request 4 would be discarded.

If j=6 is part of solution, then requests 4 and 5 would be discarded.

No: If request j is not part of solution, then solution in requests 1...j-1

Is request j part of optimal solution?

Yes: Then no request between P(j) and j can be part of optimal solution.

Optimal Substructure is when an optimal solution can be constructed from optimal solutions of sub-probs