Design and Analysis of Algorithms

2019.11.22

Lecture 02

Asymptotic Notations and Recurrence

Big-Oh

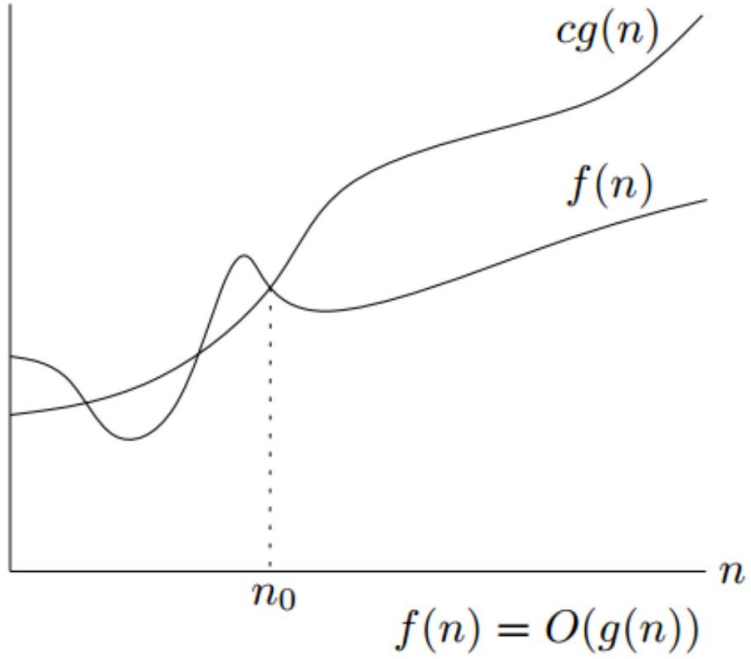
Asymptotic upper bound

Definition (big-Oh)

$f(n) = O(g(n))$: There exists constant $c > 0$ and n_0 such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$

When estimating the growth rate of $T(n)$ using big-Oh:

- ignore the low order terms
- ignore the constant coefficient of the most significant term

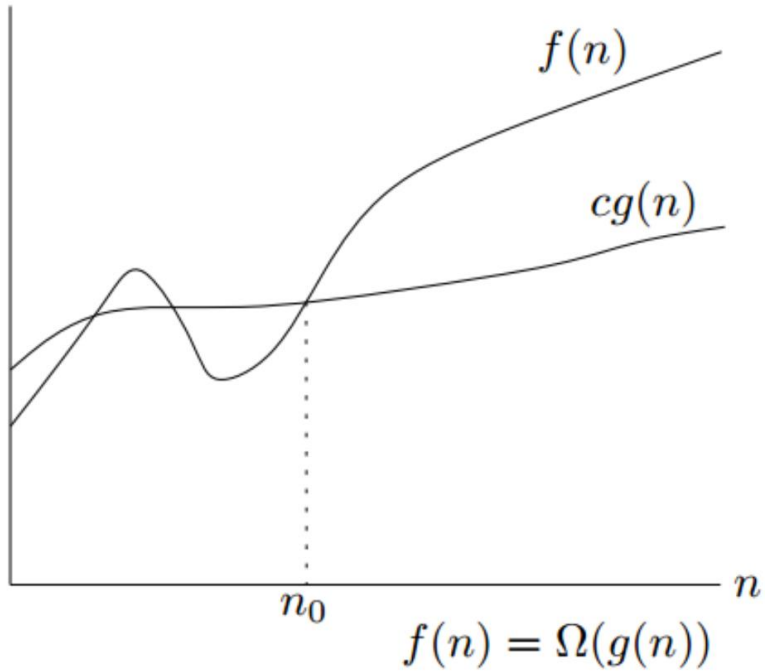


Big-Omega

Asymptotic lower bound

Definition (big-Omega)

$f(n) = \Omega(g(n))$: There exists constant $c > 0$ and n_0 such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$.



It is easy to show that

$\frac{n^2}{2} - 3n \geq \frac{n^2}{4}$ for all $n \geq 12$.

Thus, $n^2/2 - 3n = \Omega(n^2)$.

Example

$$\begin{aligned} \log(n!) &= \log(n) + \log(n-1) + \cdots + \log 1 \\ &\geq \log(n) + \log(n-1) + \cdots + \log(n/2) \\ &\geq n/2 \cdot \log(n/2) \\ &= n/2 \cdot (\log n - 1) = \Omega(n \log n). \end{aligned}$$

Asymptotic Notations

Upper bounds. T(n)=O(f(n)) if there exist constants c > 0 and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.

Equivalent definition: $\lim_{n \to \infty} \frac{T(n)}{f(n)} < \infty$.

Lower bounds. T(n)=Ω(f(n)) if there exist constants c> 0 and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) \geq c \cdot f(n)$.

Equivalent definition: $\lim_{n \to \infty} \frac{T(n)}{f(n)} > 0$.

Tight bounds. $T(n) = \Theta\big(f(n)\big)$ if $T(n) = O\big(f(n)\big)$ and $T(n) = \Omega(f(n))$.

Note: Here "=" means "is", not equal. The more mathematically correct way should be $T(n) \in O(f(n))$.

For example, for the harmonic series,

we have: $\sum_{i=1}^{n} \frac{1}{\mathrm{i}} = \mathrm{O}(\log n) = \Omega(\log n) = \Theta(\log n)$

An interesting fact about logarithm

$$\boldsymbol{log_{b_1} n = O(log_{b_2} n)}$$

For any constant $b_1 > 1$ and $b_2 > 1$.

Because of the above, in computer science, we omit all the constant logarithm bases in big-O. For example, instead of $O(\log_2 n)$, we will simply write O(log n)

- Essentially, this says that "you are welcome to put any constant base there, and it will be the same asymptotically".
- Obviously, Ω, Θ also have this property.

Substitution method: Example 1

$$T(n) = \begin{cases} 3T(n/4) + n^2, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

Prove $T(n) \leq cn^2$ by induction, where c is a large constant.

Proof.

- Base (n=1) : obviously holds for any $c \geq 1$
- Induction:

$$\begin{aligned} T(n) &= 3T(n/4) + n^2 \\ &\leq 3c(n/4)^2 + n^2 \\ &= cn^2 - (13c/16 - 1)n^2 \\ &\leq cn^2, \end{aligned}$$

whenever $13c/16 - 1 \geq 0$, or $c \geq 16/13$.

□

Proof of the Master Theorem

If $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$ for some constant a > 0, b > 1 and d ≥ 0, then

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

- The size of the subproblems decreases by a factor of b with each level of recursion, and therefore reaches the base case after $\log_b n$ levels. This is the height of the recursion tree.
- The k-th level of the tree is made up of a^k subproblems, each of size n/b^k
- The total work done at this level is

$$a^k \times O\left(\frac{n}{b^k}\right)^d = O(n^d) \times \left(\frac{a}{b^d}\right)^k$$

Lecture 03

Maximum Contiguous Subarray and Counting Inversion

The Complete Divide-and-Conquer Algorithm

$MCS(A, s, t)$

```
Input: A[s ... t] with s ≤ t
Output: MCS of A[s ... t]
begin
    if s = t then return A[s];
    else
        m ← ⌊(s+t)/2⌋;
        Find MCS(A, s, m);
        Find MCS(A, m + 1, t);
        Find MCS that contains both A[m] and A[m + 1];
        return maximum of the three sequences found
    end
end
```

First Call: *MCS*(*A, 1, n*)

Summary

In the MCS problem, we saw 3 different algorithms for solving the maximum contiguous subarray problem

- A $O(n^3)$ brute force algorithm
- A $O(n^2)$ algorithm that reuses data
- A $O(n \log n)$ divide-and-conquer algorithm

Can you solve the problem in $O(n)$ time?

Counting inversions: divide-and-conquer

- Divide: separate list into two halves A and B.
- Conquer: recursively count inversions in each list.
- Combine: count inversions (a, b) with a∈A and b∈B.
- Return sum of three counts.

Input

14	7	18	3	10	19	11	23	2	25	16	17

14	7	18	3	10	19

Count inversions in left half A

14-7,14-3,14-10,7-3,18-3,18-10

11	23	2	25	16	17

Count inversions in right half B

11-2,23-2,23-16,23-17,25-16,25-17

Count inversions (a,b) with a∈A and b∈B

14-11,14-2,7-2,18-11,18-2,18-16,18-17,3-2,10-2,19-11,19-2,19-16,19-17

Output

6+6+13 =25

Combine two subproblems: Improvement

Merge-and-Count(A, B)

```
Input: A, B
Output: r, L
r ← 0, L ← ∅;
while both A and B are not empty do
    // Let a and b represent the first element of A and B, repectively
    if a < b then
        Move a to the back of L;//A.length is decreased by 1;
    end
    else
        Increase r by A.length;
        Move b to the back of L;
    end
end
if A is not empty then
    Move A to the back of L;
end
else
    Move B to the back of L;
end
return L, r;
```

The Complete Divide-and-Conquer Algorithm

Sort-and-Count(L)

Input: L
Output: r_L, L
if L *is empty* **then**
  return $0, L$;
end
Divide L into two halves A and B;
$(r_A, A) \leftarrow$Sort-and-Count(A);//$T(\lceil \frac{n}{2} \rceil)$
$(r_B, B) \leftarrow$Sort-and-Count(B);//$T(\lfloor \frac{n}{2} \rfloor)$
$(r_L, L) \leftarrow$Merge-and-Count(A, B);//$O(n)$
return $r_A + r_B + r_L, L$;

$$T(n) = \begin{cases} O(1), & if\ n = 1 \\ T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + O(n) & otherwise \end{cases}$$

Lecture 04

Polynomial Multiplication and Quicksort

The First Divide-and-Conquer Algorithm

PolyMulti1$(A(x), B(x))$

Input: $A(x), B(x)$
Output: $A(x) \times B(x)$
$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{\frac{n}{2}}$;
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{\frac{n}{2}}$;
$U(x) \leftarrow$PolyMulti1$(A_0(x), B_0(x))$;$//T(n/2)$
$V(x) \leftarrow$PolyMulti1$(A_0(x), B_1(x))$;$//T(n/2)$
$W(x) \leftarrow$PolyMulti1$(A_1(x), B_0(x))$;$//T(n/2)$
$Z(x) \leftarrow$PolyMulti1$(A_1(x), B_1(x))$;$//T(n/2)$
return $(U(x) + [V(x) + W(x)]x^{\frac{n}{2}} + Z(x)x^n)$;$//O(n)$

$$T(n) = \begin{cases} 4T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

The improved Divide-and-Conquer Algorithm

PolyMulti2$(A(x), B(x))$

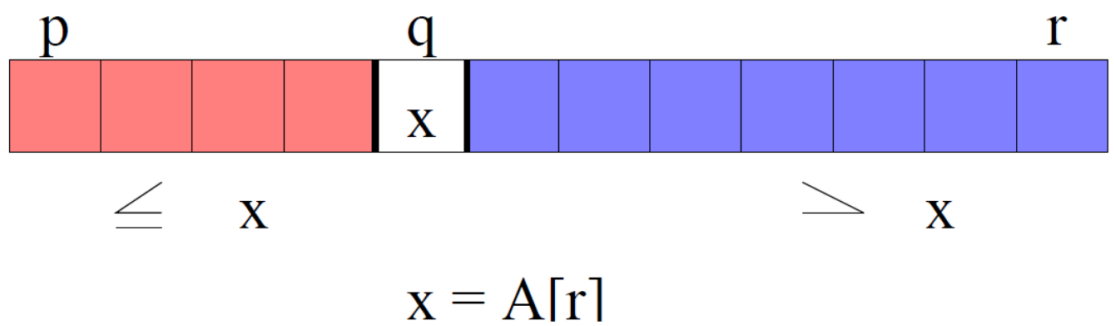
Input: $A(x), B(x)$
Output: $A(x) \times B(x)$
$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{n-\frac{n}{2}}$;
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{n-\frac{n}{2}}$;
$Y(x) \leftarrow$PolyMulti2$(A_0(x) + A_1(x), B_0(x) + B_1(x))$;$//T(n/2)$
$U(x) \leftarrow$PolyMulti2$(A_0(x), B_0(x))$;$//T(n/2)$
$Z(x) \leftarrow$PolyMulti2$(A_1(x), B_1(x))$;$//T(n/2)$
return $(U(x) + [Y(x) - U(x) - Z(x)]x^{\frac{n}{2}} + Z(x)x^{2\frac{n}{2}})$;$//O(n)$

$$T(n) = \begin{cases} 3T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

Partition

- Partition
  - Given: An array of numbers
  - Partition: Rearrange the array A[p..r] in place into two (possibly empty) subarrays A[p..q-1] and A[q+1..r] such that

$$A[u] < A[q] < A[v] \ for\ any\ p \le u \le q-1\ and\ q+1 \le v \le r$$

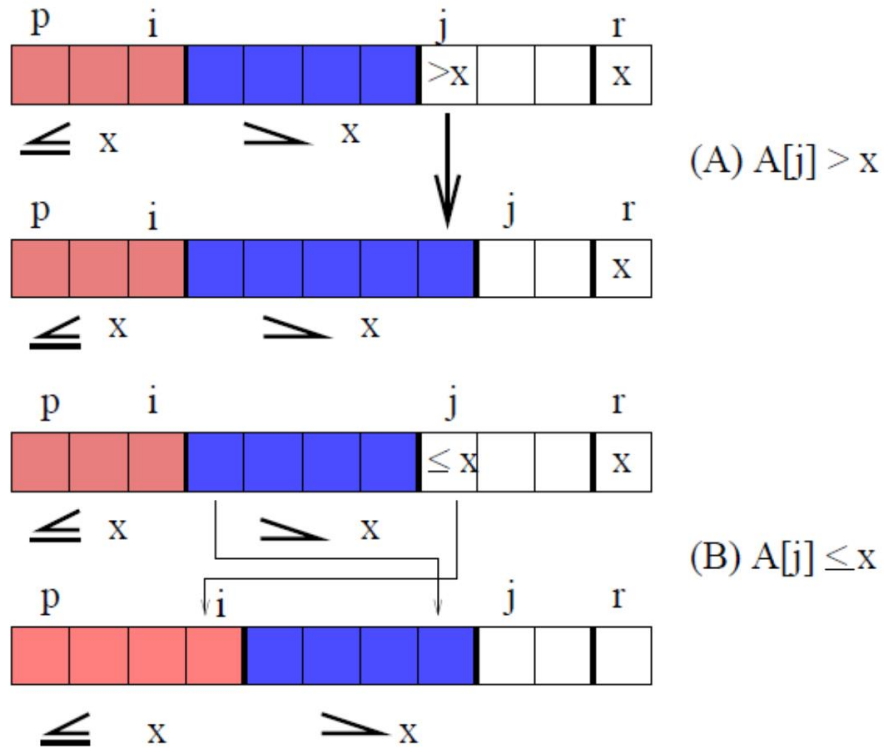


  - x is called the pivot. Assume x = A[r]; if not, swap first
  - Quicksort works by:
    - calling partition first
    - recursively sorting A[p..q-1] and A[q+1..r]

Partition

- One Iteration of the Procedure Partition
  - Increase j by 1 each time to find a place for A[j]

    At the same time increase i when necessary



  - Case (A): Only increase j by 1
  - Case (B): i = i + 1; A[i] ↔ A[j]; j = j + 1.

Partition - Pseudocode

Partition(A,p,r)

```
Input: An array A waiting to be sorted,the range of index p,r
Output: Index of the pivot after partition
x ← A[r];//A[r] is the pivot element
i ← p − 1;
for j ← p to r − 1 do
    if A[j] ≤ x then
        i ← i + 1;
        exchange A[i] and A[j];
    end
end
exchangeA[i + 1] and A[r];//Put pivot in position
return i + 1;//q ← i + 1
```

- Running time is O(r – p)
  - linear in the length of the array A[p..r]

Quicksort

Quicksort(A,p,r)

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$
Output: Sorted array $\boldsymbol{A}$
if $p < r$ **then**
  $q \leftarrow$Partition(A, p, r);
  Quicksort$(A, p, q-1)$;
  Quicksort$(A, q+1, r)$;
end
return A;

A Divide-and-Conquer Framework

- If we could always partition the array into halves, then we have the recurrence $T(n) \leq 2T(n/2) + O(n)$, hence $T(n) = O(n \log n)$.
- However, if we always get unlucky with very unbalanced partitions, then $T(n) \leq T(n-1) + O(n)$, hence $T(n) = O(n^2)$.

Randomized Algorithms

- Analysis for Randomized Algorithms:
  - Worst-case doesn't make sense: for any given input, the worst case is very unlikely to happen.
  - Use expected running time analysis for randomized algorithms!

Average case analysis

- Used for deterministic algorithms
- Assume the input is chosen randomly from some distribution
- Depends on assumptions on the input, weaker
- Not required in this course

Expected case analysis

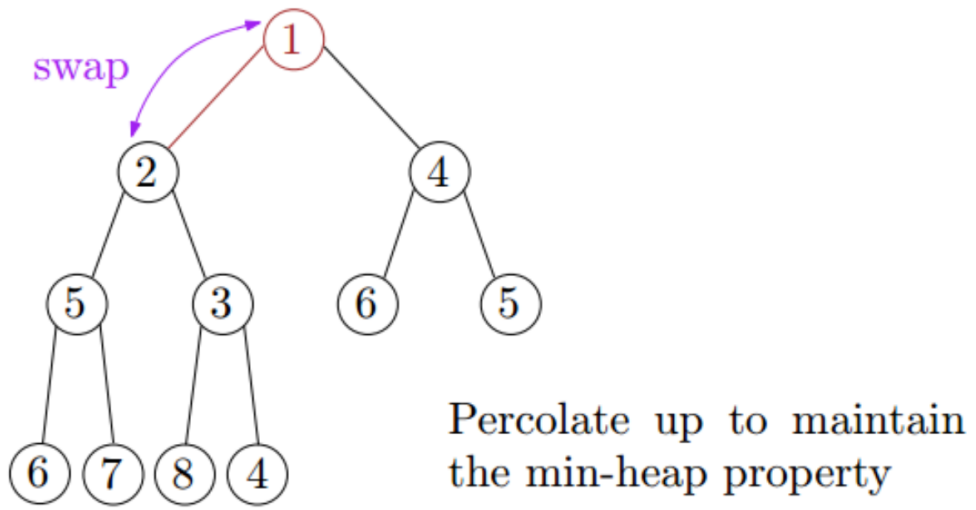
- Used for randomized algorithms
- Need to work for any given input
- Randomization is inherent within the algorithm, stronger
- Required in this course

Lecture 05

Heapsort and Lower Bound for Comparison-based Sorting

Insertion

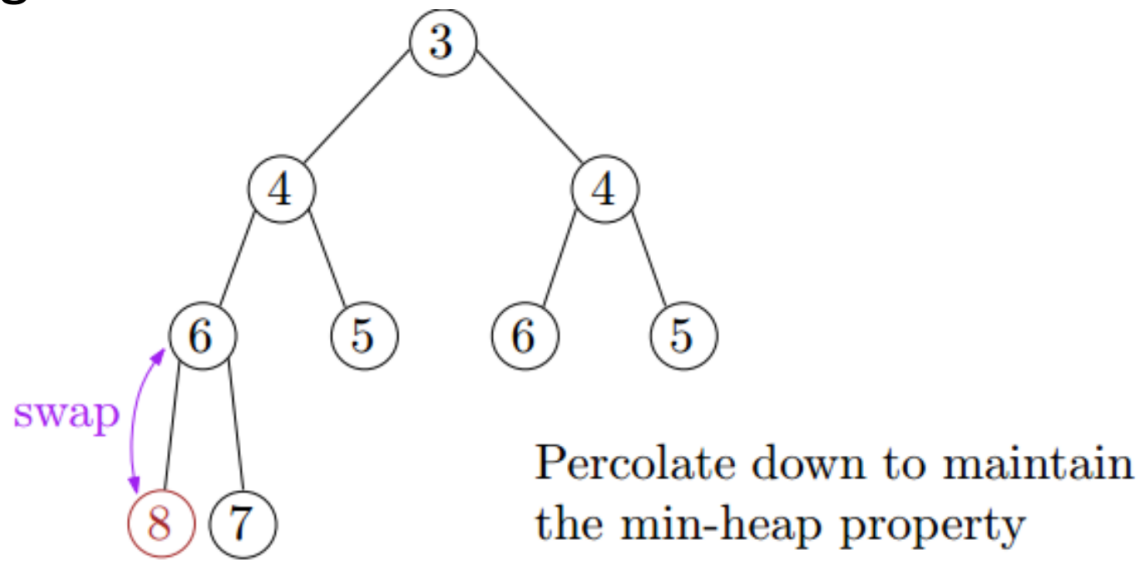
- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.



- Correctness: after each swap, the min-heap property is satisfied for the subtree rooted at the new element
- Time complexity = O(height) = O(log n)

Extract-Min

- Copy the last element to the root (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolate down (or bubble down): if the element is larger than either of its children, then interchange it with the smaller of its children.



- Correctness: after each swap, the min-heap property is satisfied for all nodes except the node containing the element (with respect to its children)
- Time complexity = O(height) = O(log n)

Heapsort

- Build a binary heap of n elements
  - the minimum element is at the top of the heap
  - insert n elements one by one
    → O(n log n)
    (there is a more efficient way, check CLRS 6.3 if interested)

- Perform n Extract-Min operations
  - the elements are extracted in sorted order
  - each Extract-Min operation takes O(log n) time
    → O(n log n)

- Total time complexity: O(n log n)

Lower Bound for Sorting

Theorem

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons.

Proof.

- A decision tree to sort n elements must have at least $n!$ leaves, since there are $n!$ possible orderings.
- A binary tree of height h has at most 2^h leaves
- Thus, $n! \leq 2^h$
  $\Rightarrow h \geq \log n! = \Omega(n \log n)$ (proved in previous lecture)

□

Corollary

Heapsort and merge sort are asymptotically optimal comparison-based sorting algorithms.

Lecture 06

0-1 Knapsack and Rod Cutting

Developing a DP Algorithm for Knapsack

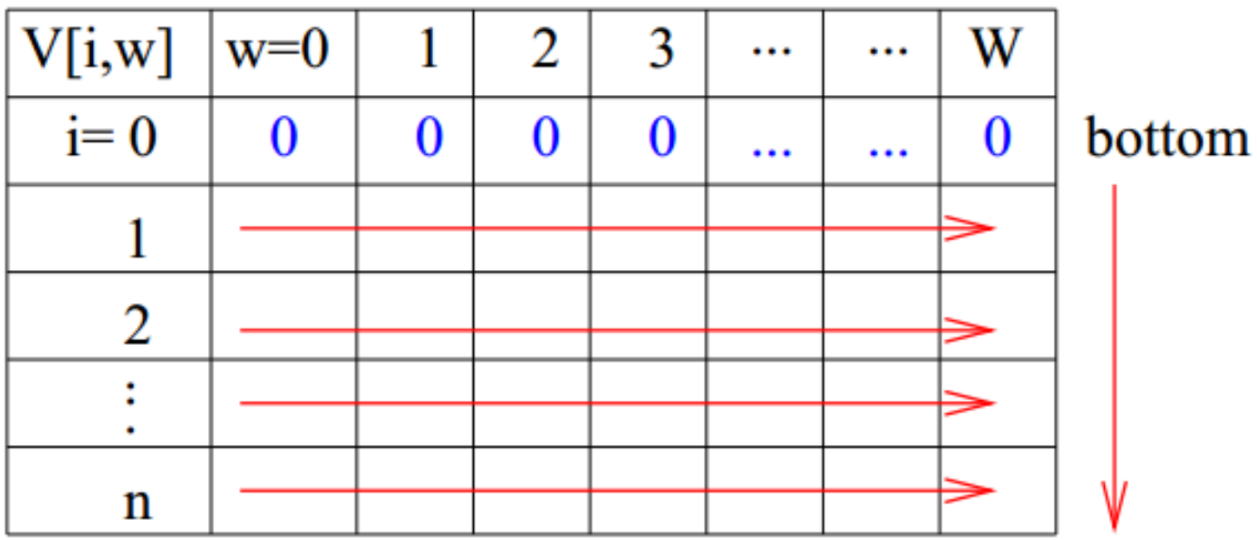
Step 3: Bottom-up computation of V [i, w]

Recurrence:

$$V[i,w] = \max(V[i-1,w], v_i + V[i-1, w-w_i])$$

We compute and save V[i, w](0 ≤ i ≤ n, 0 ≤ w ≤ W) in such an order that: When it is time to compute V [i, w], the values of V[i − 1, w] and V [i − 1, w − w_i] are available.

So, we fill the following table row by row and left to right.



The Dynamic Programming Algorithm

Knapsack(v,w,n,W)

Input: $\boldsymbol{v}$ and $\boldsymbol{w}$ are values and weights of $\boldsymbol{n}$ items, $\boldsymbol{W}$ is the allowed maximum weight of items.

Output: Maximum value of any subset of items $\{1, 2, ..., n\}$ of weight at most W.

Let $V[0..n, 0..W]$ be a new 2-dimension array;
for $w = 0$ **to** W **do**
    $V[0, w] = 0$
end
for $i = 1$ **to** n **do**
    for $w = 0$ **to** W **do**
        if $w[i] \leq w$ **then**
            $V[i, w] = \max\{V[i-1, w], v[i] + V[i-1, w - w[i]]\}$
        else
            $V[i, w] = V[i-1, w]$
        end
    end
end
return $V[n, W]$

Summary of DP

How to develop a dynamic programming? Four steps

- Structure: Analyze structure of an optimal solution, and thereby choose a space of subproblems (states).
- Recursion: Establish relationship between the optimal value the problem and those of some subproblems
- Bottom-up computation:
  - Compute the optimal values of the smallest subproblems first, save them in the table,
  - Then compute optimal values of larger subproblems, and so on, until the optimal value of the original problem is computed.
- Construction of optimal solution: Assemble optimal solution by tracing the computation at the previous step.

Notes:

- Steps 1 and 2 are related.
- Step 4 is not always necessary: we sometimes need only the optimal value.

DP Bottom-up Implementation

Bottom-Up-Cut-Rod(p,n)

```
Input: Price list p, a rod of length n.
Output: Maximum revenue q.
Let r[0..n] be a new array;// Array stores the computed optimal values.
r[0] ← 0;
for j ← 0 to n do
    // Consider problems in increasing order of size.
    q ← −∞;
    for i ← 1 to j do
        // To solve a problem of size j, we need to consider all
        decompositions into i and j − i.
        q ←max(q, p[i] + r[j − i]);
    end
    r[j] ← q;
end
return r[n];
```

- Cost: $O(n^2)$
  - The outer loop computes r[1], r[2],…, r[n] in this order
  - To compute r[j], the inner loop uses all values r[0], r[1],…, r[j − 1] (i.e., r[j − i] for $1 \le i \le j$)

Lecture 07

Chain Matrix Multiplication

Longest Common Subarray

Minimum Editing Distance

A Dynamic Programming Algorithm

Step 3: Bottom-up computation of m[i, j]

Recurrence:

$$m[i,j] = \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1} p_k p_j)$$

Compute and save *m*[*i, j*] in such an order that when it is time to calculate *m*[*i, j*], the values of *m*[i, *k*] and *m*[*k* + 1, *j*] are already available.

Compute them in increasing order of the length of the matrix-chain:

m[1,2], m[2,3], m[3,4],…, m[n-3,n-2], m[n-2,n-1], m[n-1,n]
m[1,3], m[2,4], m[3,5],…, m[n-3,n-1], m[n-2,n]
m[1,4], m[2,5], m[3,6],…, m[n-3,n]…
m[1,n-1], m[2,n]
m[1,n]

The Dynamic Programming Algorithm

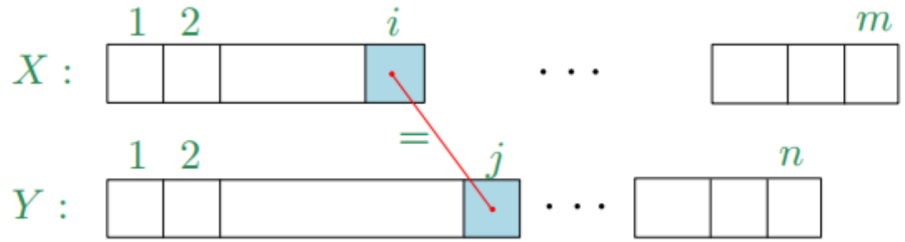
MatrixChain(p,n)

```
Let m[1..n, 1..n] and s[1..n, 1..n] be two 2-dimension arrays;
for i ← 1 to n do
|  m[i, i] ← 0;
end
for l ← 2 to n do
|  for i ← 1 to n − l + 1 do
|  |  j ← i + l − 1;
|  |  m[i, j] ← ∞;
|  |  for k ← i to j − 1 do
|  |  |  q ← m[i, k] + m[k + 1, j] + p[i − 1] ∗ p[k] ∗ p[j];
|  |  |  if q < m[i, j] then
|  |  |  |  m[i, j] ← q;
|  |  |  |  s[i, j] ← k;
|  |  |  end
|  |  end
|  end
end
return m[1, n] and s;
```

Complexity: The loops are nested three levels deep. Each loop index takes on $\leq n$ values. Hence the time complexity is $O(n^3)$. Space complexity is $\Theta(n^2)$.

Relationships among Subproblems

Step 2: Relating the value of a problem and those of its subproblems



Let $Z_k = (z_1, \ldots, z_k)$ be a LCS of X[1..i] and Y [1..j].

Case 1: If $x_i = y_j$, then $z_k = x_i = y_j$ and Z_{k-1} is a LCS of X[1..i−1] and Y[1..j−1].

Case 2: If $x_i \neq y_j$, it means that either the LCS does not end with x_i or does not end with y_j. Then Z_k is either a LCS of X[1..i−1] and Y[1..j], or a LCS of X[1..i] and Y [1..j−1]. We carry on with the larger LCS count from either case.

$$d_{i,j} = \begin{cases} d_{i-1,j-1} + 1, & if\ x_i = y_j \\ max\{d_{i-1,j}, d_{i,j-1}\}, & if\ x_i \neq y_j \end{cases}$$

The Dynamic Programming Algorithm

```
//Dynamic Programming
for i ← 1 to m do
    for j ← 1 to n do
        if x_i is equal to y_j then
            d[i, j] ← d[i − 1, j − 1] + 1;
            p[i, j] ←"LU";//"LU" indicates left up arrow.
        end
        else if d[i − 1, j] ≥ d[i, j − 1] then
            d[i, j] ← d[i − 1, j];
            p[i, j] ←"U";//"U" indicates up arrow.
        end
        else
            d[i, j] ← d[i, j − 1];
            p[i, j] ←"L";//"L" indicates left arrow.
        end
    end
end
return d, p;
```

Obviously, the dynamic programming algorithm runs in O(mn) time.

Relationships among Subproblems

Step 2: Relating the value of a problem and those of its subproblems

Case 1: If $x_i = y_j$, then $z_k = x_i = y_j$ and Z_{k-1} is a LCS of X and Y ending at x_{i-1} and y_{j-1}

Case 2: If $x_i \neq y_j$, then there can't be a common substring ending at x_i and y_j

$$d_{i,j} = \begin{cases} d_{i-1,j-1} + 1, & if\ x_i = y_j \\ 0, & if\ x_i \neq y_j \end{cases}$$

Finally, we can have the longest common substring by finding the maximum of what we have computed for all possible ending positions i and j.

$$\text{LCSubString}(X, Y) = \max\{d_{i,j}\}$$

The Dynamic Programming Algorithm

```
//Dynamic Programming
for i ← 1 to m do
    for j ← 1 to n do
        if x_i ≠ y_j then
            d[i, j] ← 0;
        end
        else
            d[i, j] ← d[i − 1, j − 1] + 1;
            if d[i, j] > l_max then
                l_max ← d[i, j];
                p_max ← i;
            end
        end
    end
end
return l_max, p_max;
```

Relationships among Subproblems

Step 2: Relating the value of a problem and those of its subproblems

To turn X[1..i] into Y[1..j], we have three cases:

Case 1: Turn X[1..i-1] into Y[1..j] and delete X[i]

Example 1 : MED (cxy -> dab) = MED (cx -> dab) + 1

Case 2: Turn X[1..i] into Y[1..j-1] and insert Y[j]

Example 2 : MED (cxy -> dab) = MED (cxy -> da) + 1

Case 3: Turn X[1..i-1] into Y[1..j-1] and substitute X[i] with Y[j] if needed (X[i] ≠ Y[j])

Example 3.1 : MED (cxy -> dab) = MED (cx -> da) + 1

Example 3.2 : MED (cxy -> day) = MED (cx -> da)

$$D[i,j] = min \begin{cases} D[i-1,j]+1 \\ D[i,j-1]+1 \\ D[i-1,j-1] + \begin{cases} 0 & if\ X[i] = Y[j] \\ 1 & otherwise. \end{cases} \end{cases}$$

The Dynamic Programming Algorithm

Minimum-Edit-Distance(X,Y)

Input: Two strings $\boldsymbol{X}$,$\boldsymbol{Y}$.
Output: Minimum edit distance of X and Y.
$m \leftarrow$length(X);
$n \leftarrow$length(Y);
Let $d[0..m, 0..n]$ and $p[0..m, 0..n]$ be two new 2-dimension arrays;
//Initializaiton
for $i \leftarrow 0$ *to* m **do**
  $d[i, 0] \leftarrow i$;
  $p[i, 0] \leftarrow$"U";
end
for $j \leftarrow 0$ *to* n **do**
  $d[0, j] \leftarrow j$;
  $p[0, j] \leftarrow$"L";
end

The Dynamic Programming Algorithm

```
//Dynamic Programming
for i ← 1 to m do
    for j ← 1 to n do
        if x_i is not equal to y_j then
            c ← 1;
        end
        else
            c ← 0;
        end
        if d[i − 1, j − 1] + c ≤ d[i − 1][j] + 1 and
         d[i − 1, j − 1] + c ≤ d[i][j − 1] + 1 then
            d[i, j] ← d[i − 1, j − 1] + c;
            p[i, j] ←"LU";//"LU" indicates left up arrow.
        end
        else if d[i, j − 1] + 1 < d[i − 1][j] + 1 and
         d[i, j − 1] + 1 < d[i − 1, j − 1] + c then
            d[i, j] ← d[i, j − 1] + 1;
            p[i, j] ←"L";//"L" indicates up arrow.
        end
        else
            d[i, j] ← d[i − 1, j] + 1;
            p[i, j] ←"U";//"U" indicates left arrow.
        end
    end
end
return d, p;
```

Lecture 08

Midterm Review

Analysis

Counting-Sort(A,B,k)

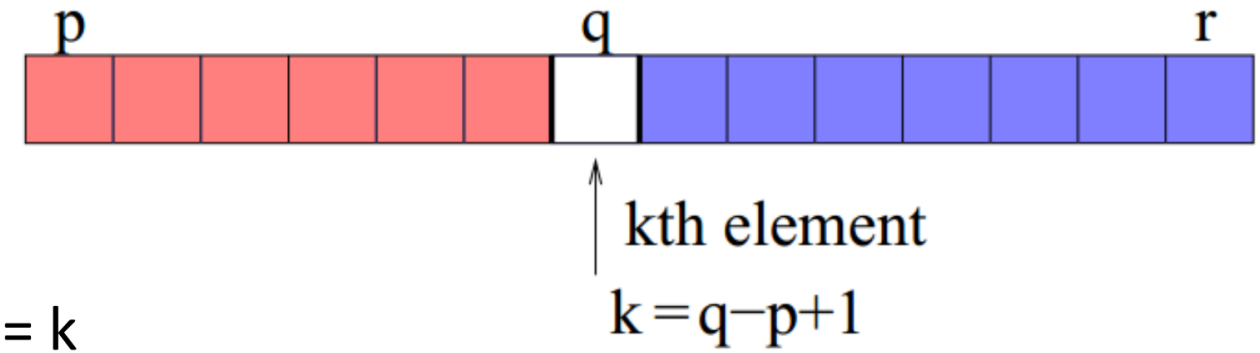
Input: $\boldsymbol{A}[1...n]$ where $A[j] \in \{1, 2, ..., k\}$
Output: $\boldsymbol{B}[1...n]$,sorted
let $C[1...k]$ be a new array;
for $i \leftarrow 1$ *to* k **do**
  $C[i] \leftarrow 0$;//$O(k)$
end
for $j \leftarrow 1$ *to* n **do**
  $C[A[j]] \leftarrow C[A[j]] + 1$;//$O(n)$
end
for $i \leftarrow 2$ *to* k **do**
  $C[i] \leftarrow C[i] + C[i-1]$;//$O(k)$
end
for $j \leftarrow n$ *to* 1 **do**
  $B[C[A[j]]] \leftarrow A[j]$;
  $C[A[j]] \leftarrow C[A[j]] - 1$;//$O(n)$
end
return B;

Total: $O(n+k)$

Randomized-Select(A,p,r,i), 1≤i≤r−p+1

Problem: Select the *i*th smallest element in A[p..r], where $1 \le i \le r-p+1$

Solution: Apply Randomized-Partition(A, p, r), getting



- i = k
  - pivot is the solution
- i < k
  - the *i*th smallest element in A[p..r] must be the *i*th smallest element in A[p..q−1]
- i > k
  - the *i*th smallest element in A[p..r] must be the (*i* − *k*)th smallest element in A[q+1..r]

If necessary, recursively call the same procedure to the subarray

Randomized-Select(A,p,r,i), 1≤i≤r−p+1

Randomized-Select(A,p,r,i)

Input: An array $\boldsymbol{A}$,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$,the $\boldsymbol{i}$th smallest element that we want to select
Output: The ith smallest element $\boldsymbol{A}[\boldsymbol{i}]$
if p *is equal to* r **then**
  return $A[p]$;
end
$q \leftarrow$Randomized-Partition(A, p, r);
$k \leftarrow q - p + 1$;
if $i \leftarrow k$ **then**
  return $A[q]$;//The pivot is the answer
end
else if $i < k$ **then**
  return Randomized-Select$(A, p, q - 1, i)$;
end
else
  return Randomized-Select$(A, q + 1, r, i - k)$;
end

To find the ith smallest element in A[1..n], call Randomized-Select(A, 1, n, i)

Randomized Quicksort vs Randomized Selection

Question

Why does Randomized Selection take $O(n)$ time while Randomized Quicksort takes $O(n \log n)$ time?

Answer:

- Randomized Selection needs to work on only 1 of the two subproblems.
- Randomized Quicksort needs to work on both of the two subproblems.

A DP Algorithm for Optimal BST Problem

Step 2: Relating the value of a problem and those of its subproblems

- Choose k_r as the root that gives the lowest expected search cost, giving us our final recursive formulation:

$$e[i,j] = \begin{cases} q_{i-1} & if\ j = i-1, \\ \min\limits_{i \le r \le j} \{ e[i, r-1] + e[r+1, j] + w[i,j] \} & if\ i \le j. \end{cases}$$

- *e[i, j]* give the expected search costs in optimal BST.

Lecture 09

Greedy Algorithm

Huffman Encoding Problem

- $d(a_i)$, the depth of leaf a_i, is equal to the length $L(c(a_i))$ of the codeword associated with that leaf.
- Weighted external pathlength B(T) of tree T

$$\sum_{i=1}^{n} f(a_i)d(a_i) = \sum_{i=1}^{n} f(a_i)L(c(a_i))$$

Definition (Minimum-Weight External Pathlength Problem)

Given weights $f(a_1), \ldots, f(a_n)$, find a tree T with n leaves labeled $a_1, \ldots, a_n$ that has minimum weighted external path length.

The Huffman encoding problem is equivalent to the minimum weighted external path length problem.

Huffman Coding Algorithm

Given an alphabet A with frequency distribution {f(a):a∈A}. The binary Huffman tree is constructed using a priority queue, Q, of nodes, with frequencies as keys.

Huffman(A)

Input: An alphabet $\boldsymbol{A}$ with frequency distribution.
Output: Huffman tree.
$n \leftarrow |A|$;
$Q \leftarrow$ a new Priority Queue of A;
for $i \leftarrow 1$ *to* $n-1$ **do**
  // Why $n-1$?
  $z \leftarrow$ a new node;
  $z.left \leftarrow$Extract-Min(Q);
  $z.right \leftarrow$Extract-Min(Q);
  $z.freq \leftarrow z.left.freq + z.right.freq$;
  Insert(Q, z);
end
return *Extract-Min(Q)*;

Running time is O(n log n), as each priority queue operation takes time O(log n).

Pseudocode

Greedy Activity Selection(A)

```
Input: a set of activities A = a_1, a_2 ..., a_n
Output: the largest subset of A that do not overlap
Sort activities in increasing order of finishing time;  // cost: O(n log n)
P = a_1; // insert the activity with earliest finishing time
k = 1; // index to the last activity in A
for i ← 2 to n do                                       // cost: O(n)
    if s[i] ≥ f[k] then
        // i starts after k finishes - no overlap
        P ← P ∪ a_i;
        k ← i;
    end
end
return P;
```

- Total Time Complexity: $O(n \log n)$

Correctness

- Proving a greedy solution is optimal:
  - Remember: Not all problems have optimal greedy solution.
  - If it does, you need to prove it.
  - Usually the proof includes mapping or converting any other optimal solution to the greedy solution.

A Dynamic Programming Algorithm

Step 2: Relating a problem to its subproblems

- Case 1. $OPT(j)$ does not select activity a_j.
  - Must be an optimal solution to problem consisting of remaining activities $a_1, a_2, \cdots, a_{j-1}$
- Case 2. $OPT(j)$ selects activity a_j.
  - Collect profit w_j
  - Can't use incompatible activities $\{a_{p(j)+1}, a_{p(j)+2}, \cdots, a_{j-1}\}$
  - Must include optimal solution to problem consisting of remaining compatible activities $a_1, a_2, \cdots, a_{p(j)}$
- Dynamic Programming Equation:

$$OPT(j) = \begin{cases} 0 & if\ j = 0 \\ \max\{OPT(j-1), w_j + OPT(p(j))\} & if\ j > 0 \end{cases}$$

Lecture 10

BFS/DFS

Topological Sort

SCCs

BFS Algorithm

BFSVisit(s)

Input: A vertex s
Output: None
$color[s] \leftarrow$ GRAY,$d[s] \leftarrow 0$;
$Q \leftarrow \emptyset$,Enqueue(Q, s);
while $Q \neq \emptyset$ **do**
  $u \leftarrow$Dequeue(Q);
  for $v \in Adj[u]$ **do**
    if $color[v] \leftarrow WHITE$ **then**
      $color[v] \leftarrow$GRAY;
      $d[v] \leftarrow d[u] + 1$;
      $pred[v] \leftarrow u$;
      Enqueue(Q, v);
    end
  end
  $color[u] \leftarrow$BLACK;
end

DFS Algorithm

DFSVisit(u)

```
Input: A vertex u
Output: None
color[u] ← GRAY;//u is discovered
d[u] ← ++time;//u's discovery time
for v ∈ Adj(u) do
    //Visit undiscovered vertex
    if color[v] is equal to WHITE then
        pred[v] ← u;
        DFSVisit(v);
    end
end
color[u] ←BLACK;//u has finished
f[u] ← ++time;//u's finish time
```

Topological Sort Algorithm

Topological-Sort(G)

```
Input: A graph G
Output: None
Initialize Q to be an empty queue;
for u ∈ V do
    if u.in_degree is equal to 0 then
        //Find all starting vertices
        Enqueue(Q, u);
    end
end
while Q is not empty do
    u ←Dequeue(Q);
    Output u;
    for v ∈ Adj(u) do
        //remove u's outgoing edges
        v.in_degree ← v.in_degree − 1;
        if v.in_degree is equal to 0 then
            Enqueue(Q, v);
        end
    end
end
```

Strongly Connected Components

Let $G = (V, E)$ be a directed graph.

A strongly connected component (SCC) of G is a subset S of V such that

- For any two vertices u, $v \in S$, it must hold that:
  - There is a path from u to v.
  - There is a path from v to u.
- S is maximal in the sense that we cannot put any more vertex into S without violating the above property.

Algorithm

Step 1: Obtain the reverse graph G^R by reversing the directions of all the edges in G.

Step 2: Perform DFS on G^R, and obtain the sequence L^R that the vertices in G^R turn black (i.e., whenever a vertex is popped out of the stack, append it to L^R).

Obtain L as the reverse order of L^R.

Step 3: Perform DFS on the original graph G by obeying the following rules:

- **Rule 1**: Start the DFS at the first vertex of L.
- **Rule 2**: Whenever a restart is needed, start from the first vertex of L that is still white.

Output the vertices in each DFS-tree as an SCC.

Pseudocode

SCC(G)

```
Input: A directed graph G
Output: The set of strongly connected components R
R ← {};// set of SCCs
G^R ←reverse graph of G;
L^R ←DFS-b(G^R);// Perform DFS
L ←reverse order of L^R;
for u ∈ L do
    if color[u] is equal to WHITE then
        Lscc ←DFSVisit(G,u);// Perform DFS starting at u
        R ← R∪Set(Lscc);
    end
end
return R;
```

SCC Graph

Let G be the input directed graph, with SCCs $S_1, S_2, \ldots, S_t$ for some t ≥ 1.

Let us define a SCC graph G^{SCC} as follows:
Each vertex in G^{SCC} is a distinct SCC in G.
Consider two vertices (a.k.a. SCCs) S_i and S_j ($1 \le i,j \le t$).
G^{SCC} has an edge from S_i to S_j if and only if

- $i \neq j$, and
- There is a path in G from a vertex in S_i to a vertex in S_j.

Finding SCCs - The Strategy

The previous lemma suggests the following strategy for finding all the SCCs:

- Performing DFS from any vertex in a sink SCC S.
- Delete all the vertices of S from G, as well as their edges.
- Accordingly, delete S from G^{SCC}, as well as its edges.
- Repeat from Step 1, until G is empty.

Lecture 11

Minimum Spanning Tree

How to Find a Safe Edge?

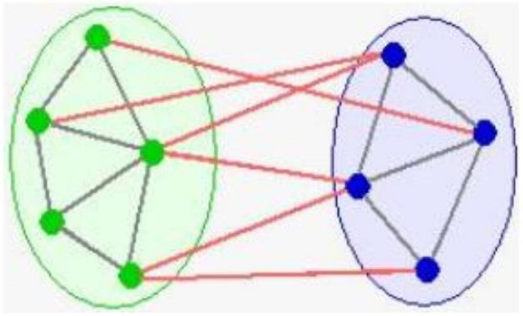
Lemma

- *Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E*
- *A be a subset of E that is included in some minimum spanning tree for G.*

Let

- *$(S, V - S)$ be any cut of G that respects A*
- *(u, v) be a light edge crossing the cut $(S, V - S)$*

Then, edge (u, v) is safe for A.



It means that we can find a safe edge by

1. first finding a cut that respects A,
2. then finding the light edge crossing that cut.

That light edge is a safe edge.

Description of Prim's Algorithm

Prim(G,w,r)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, the algorithm will start at root vertex $\boldsymbol{r}$
Output: None
Let $color[1...|V|], key[1...|V|], pred[1...|V|]$ be new arrays;
for $u \in V$ **do**
  $color[u] \leftarrow$ WHITE,$key[u] \leftarrow +\infty$;// Initialize
end
$key[r] \leftarrow 0$, $pred[r] \leftarrow$ NULL;// Start at root vertex
$Q \leftarrow$ new PriQueue(V);// put vertices in Q
while Q *is nonempty* **do**
  $u \leftarrow Q$.Extract-Min();// lightest edge
  for $v \in adj[u]$ **do**
    if $(color[v] \leftarrow WHITE)\&\&(w[u,v] < key[v])$ **then**
      $key[v] \leftarrow w[u,v]$;// new lightest edge
      Q.Decrease-Key$(v, key[v])$;
      $pred[v] \leftarrow u$;
    end
  end
  $color[u] \leftarrow$ BLACK;
end

Analysis of Prim's Algorithm

The data structure PriQueue (heap) supports the following two operations: (See CLRS)

- O(log V) for Extract-Min on a PriQueue of size at most V .
  Total cost: O(V log V)
- O(log V) time for Decrease-Key on a PriQueue of size at most E.
  Total cost: O(E log V).

Total cost is then O((V + E) log V) = O(E log V)

Idea of Kruskal's Algorithm

- Kruskal's Algorithm is based directly on the generic algorithm.
- Unlike Prim's algorithm, which grows one tree, Kruskal's algorithm grows a collection of trees (a forest).
- Initially, this forest consists of the vertices only (no edges).
- In each step the cheapest edge that does not create a cycle is added.
- Continue until the forest 'merges into' a single tree.

How to Check for Cycles?

Observations:

- At each step of the framework algorithm, (V,A) is acyclic so it is a forest.
- If u and v are in the same tree, then adding edge {u,v} to A creates a cycle.
- If u and v are not in the same tree, then adding edge {u,v} to A does not create a cycle.

Question

How to test whether u and v are in the same tree?

High-Level Answer: Use a disjoint-set data structure

- Vertices in a tree are considered to be in same set.
- Test if Find-Set(u) = Find-Set(v)?

Low-Level Answer:

- The Union-Find data structure implements this

Time Complexity of Kruskal's Algorithm

Input: A graph G, a matrix w representing the weights between vertices in G

Output: MST of G

Sort E in increasing order by weight w;// $O(|E| \log |E|)$

// After sorting $E = \langle \{u_1, v_1\}, \{u_2, v_2\}, ..., \{u_{|E|}, v_{|E|}\} \rangle$

$A \leftarrow \{\}$;

for $u \in V$ **do**

  Create-Set(u);// $O(|V|)$

end

for $e_i \in E$ **do**

  // $O(|E| \log |V|)$

  if *Find-Set*(u_i)$\neq$*Find-Set*(v_i) **then**

    add $\{u_i, v_i\}$ to A;

    Union(u_i, v_i);

  end

end

return A;

Remark: With a proper implementation of Union-Find, Kruskal's algorithm has running time $O(|E| \log |E|) = O(|E| \log |V|)$.

Lecture 12

Single Source Shortest Pace

Description of Dijkstra's Algorithm

Dijkstra(G,w,s)

```
Input: A graph G, a matrix w representing the weights between vertices
       in G, source vertex s
Output: None
for u ∈ V do
  | d[u] ← ∞, color[u] ← WHITE;// Initialize
end
d[s] ← 0;
pred[s] ← NULL;
Q ←queue with all vertices;
while Non-Empty(Q) do
  | // Process all vertices
  | u ←Extract-Min(Q);// Find new vertex
  | for v ∈ Adj[u] do
  |   | if d[u] + w(u, v) < d[v] then
  |   |   | // If estimate improves
  |   |   | d[v] ← d[u] + w(u, v);// relax
  |   |   | Decrease-Key(Q, v, d[v]);
  |   |   | pred[v] ← u;
  |   | end
  | end
  | color[u] ←BLACK;
end
```

Analysis of Dijkstra's Algorithm

- The initialization uses only O(|V|) time.
- Each vertex is processed exactly once, so Non-Empty() and Extract-Min() are called exactly once, i.e., O(|V|) times in total.
- The inner loop for (each v ∈ Adj[u]) is called once for each edge in the graph. Each call of the inner loop does O(1) work plus, possibly, one Decrease-Key operation.
- Recalling that all of the priority queue operations require O(log |Q|) = O(log|V|) time, we have that the algorithm uses

$$O(|V|) + |E| \cdot O(1 + \log|V|) = O(|E| \log |V|)$$

time.

Description of Bellman-Ford Algorithm

- The algorithm relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$.

Bellman-Ford(G,w,s)

Input: A directed graph $\boldsymbol{G}$,weights $\boldsymbol{w}$, and the source vertex $\boldsymbol{s}$
Output: Return FALSE if G contains negative cycle, return TRUE if shortest paths from s to any other vertices obtained.

```
for u ∈ V do
|  d[u] ← ∞, pred[u] ←NIL;// Initialize
end
for i ← 1 to |V| − 1 do
|  for e ∈ E do
|  |  RELAX(u, v, w);
|  end
end
for e ∈ E do
|  if d[v] > d[u] + w(u, v) then
|  |  return FALSE;
|  end
end
return TRUE;
```

Analysis of Bellman-Ford Algorithm

- The Bellman-Ford algorithm runs in time $O(|V| \cdot |E|)$ since the initialization takes $O(|V|)$ time, each of the $|V| - 1$ passes over the edges takes $O(|E|)$ time, and the **for** loop takes $O(|E|)$ time.

Bellman-Ford(G,w,s)

Input: A directed graph G,weights w, and the source vertex s
Output: Return FALSE if G contains negative cycle, return TRUE if shortest paths from s to any other vertices obtained.

```
for u ∈ V do
|  d[u] ← ∞, pred[u] ←NIL;// Initialize
end
for i ← 1 to |V| − 1 do
|  for e ∈ E do
|  |  RELAX(u, v, w);
|  end
end
for e ∈ E do
|  if d[v] > d[u] + w(u, v) then
|  |  return FALSE;
|  end
end
return TRUE;
```

$O(|V|)$

$O(|V| \cdot |E|)$

$O(|E|)$

Lecture 13

Dealing with Hard Problems

Optimization and Decision Problems

- For almost all optimization problems there exists a corresponding simpler decision problem.
- Given a subroutine for solving the optimization problem, solving the corresponding decision problem is usually trivial.

Example

If we know how to solve MST, we can solve DST which asks if there is an Spanning Tree with weight at most k.
How? First solve the MST problem and then check if the MST has cost $\leq k$. If it does, answer Yes. If it doesn't, answer No.

- Thus if we prove that a given decision problem is hard to solve efficiently, then it is obvious that the optimization problem must be (at least as) hard.

Note: It will be more convenient to compare the 'hardness' of decision problems than of optimization problems (since all decision problems share the same form of output, either yes or no.)

Polynomial-Time Algorithms

Definition

An algorithm is polynomial-time if its running time is $O(n^k)$, where k is a constant independent of n, and n is the input size of the problem that the algorithm solves.

- Remark: Whether you use n or n^α (for fixed a > 0) as the input size, it will not affect the conclusion of whether an algorithm is polynomial time.
  - This explains why we introduced the concept of two functions being of the same type earlier on.
  - Using the definition of polynomial-time it is not necessary to fixate on the input size as being the exact minimum number of bits needed to encode the input!

The Class P

Definition

The class $\mathcal{P}$ consists of all decision problems that are solvable in polynomial time. That is, there exists an algorithm that will decide in polynomial time if any given input is a yes-input or a no-input.

Question

How to prove that a decision problem is in $\mathcal{P}$?

Ans: You need to find a polynomial-time algorithm for this problem.

Question

How to prove that a decision problem is not in $\mathcal{P}$?

Ans: You need to prove there is no polynomial-time algorithm for this problem (much harder).

Certificates and Verifying Certificates

Definition

A Certificate is a specific object corresponding to a yes-input, such that it can be used to show that the input is indeed a yes-input.

- By definition, only yes-input needs a certificate (a no-input does not need to have a 'certificate' to show it is a no-input).

- Verifying a certificate: Given a presumed yes-input and its corresponding certificate, by making use of the given certificate, we verify that the input is actually a yes-input.

The Class NP

Definition

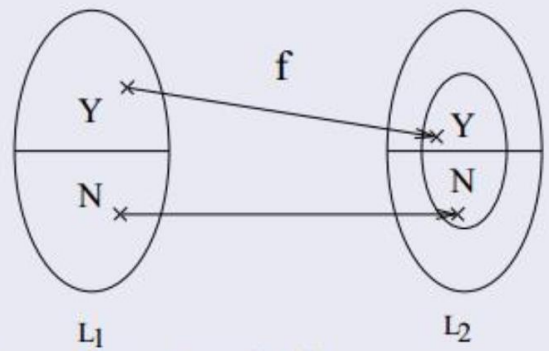
The class $\mathcal{NP}$ consists of all decision problems such that, for each yes-input, there exists a certificate which allows one to verify in polynomial time that the input is indeed a yes-input.

- Remark: NP stands for "nondeterministic polynomial time". The class NP was originally studied in the context of nondeterminism, here we use an equivalent notion of verification.

Polynomial-Time Reductions

Definition

- Let L_1 and L_2 be two decision problems.
- A Polynomial-Time Reduction from L_1 to L_2 is a transformation f with the following two properties:
  1. f transforms an input x for L_1 into an input $f(x)$ for L_2 such that
     - a yes-input of L_1 maps to a yes-input of L_2, and a no-input of L_1 maps to a no-input of L_2.



  2. $f(x)$ is computable in polynomial time (in $size(x)$).

If such an f exists, we say that L_1 is polynomial-time reducible to L_2, and write $L_1 \leq_P L_2$.

The Class NP-Complete (NPC)

- We have finally reached our goal of introducing class NPC.

Definition

The class $\mathcal{NPC}$ of $\mathcal{NP}$-complete problems consists of all decision problems L such that

1. $L \in \mathcal{NP}$;
2. for every $L' \in \mathcal{NP}$, $L' \leq_P L$.

- Intuitively, NPC consists of all the hardest problems in NP.

Cook's Theorem (SAT $\in$ NPC)

Question

How do we prove one problem in $\mathcal{NPC}$ to start with?

Theorem (Cook's Theorem (1971))

$\text{SAT} \in \mathcal{NPC}$.

- Remark: Since Cook showed that SAT $\in$ NPC, thousands of problems have been shown to be in NPC using the reduction approach described earlier.
- Remark: With a little more work we can also show that 3-SAT $\in$ NPC as well.
- Note: For the purposes of this course you only need to know the validity of Cook's Theorem, and 3-SAT $\in$ NPC but do not need to know how to prove them.

Decision versus Optimization Problems

In General,

- If a decision problem can be solved in polynomial time, then the corresponding optimization problems can also be solved in polynomial time.

- If a decision problem cannot be solved in polynomial time, then the corresponding optimization problems cannot be solved in polynomial time either.

NP-Hard Problems

Definition

A problem L is $\mathcal{NP}$-hard if problem in $\mathcal{NPC}$ can be polynomially reduced to it (but L does not need to be in $\mathcal{NP}$).

- In general, the optimization versions of NP-Complete problems are NP-Hard.

Example

VC: Given an undirected graph G, find a minimum-size vertex cover.
DVC: Given an undirected graph G and k, is there a vertex cover of size k?
If we can solve the optimization problem VC, we can easily solve the decision problem DVC.

- Simply run VC on graph G and find a minimum vertex cover S.
- Now, given (G, k), solve $DVC(G, k)$ by checking whether $k \geq |S|$. If $k \geq |S|$, answer Yes, if not, answer No.