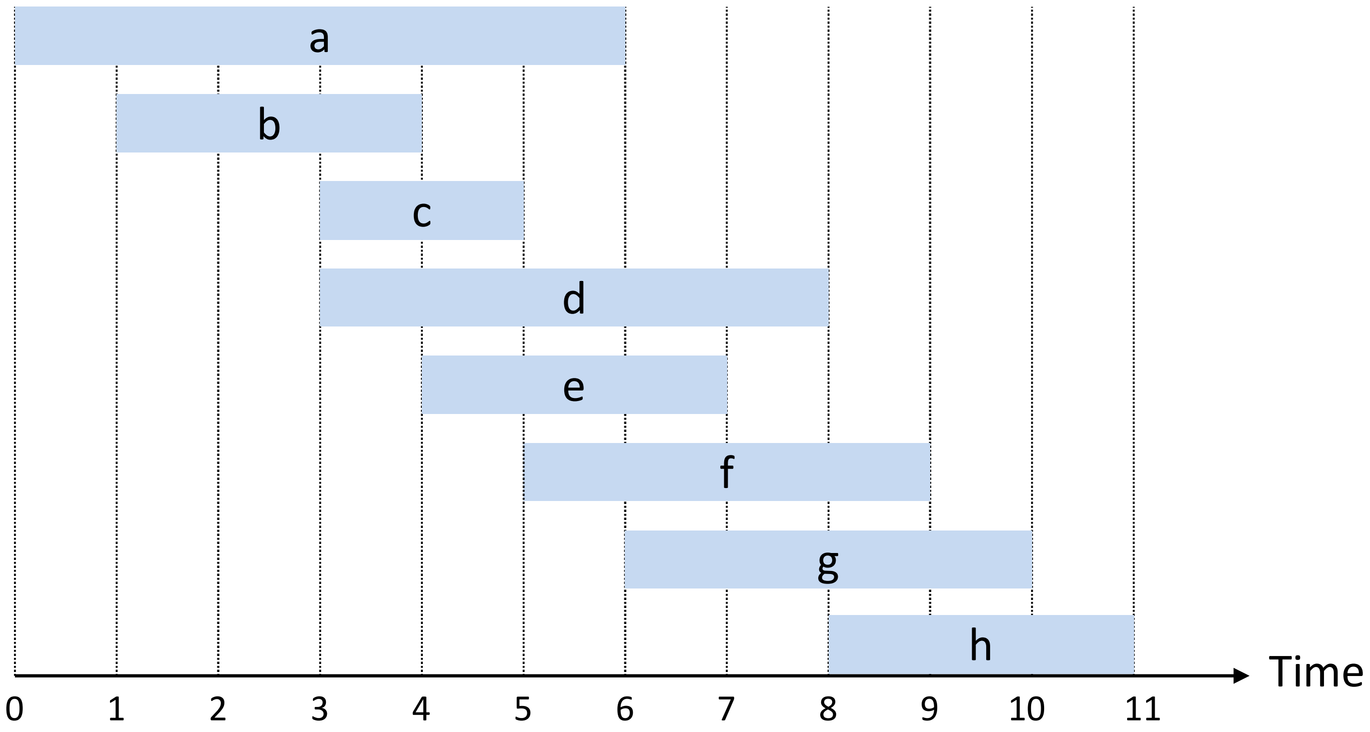
Revision

Greedy Algorithm

Interval Scheduling

- Interval scheduling.
  - Job j starts at s_j and finishes at f_j.
  - Two jobs compatible if they don't overlap.
  - Goal: find maximum subset of mutually compatible jobs.



Interval Scheduling: Greedy Algorithm

- Greedy algorithm. Consider jobs in increasing order of finishing time. Take each job provided it's compatible with the ones already taken.

jobs selected

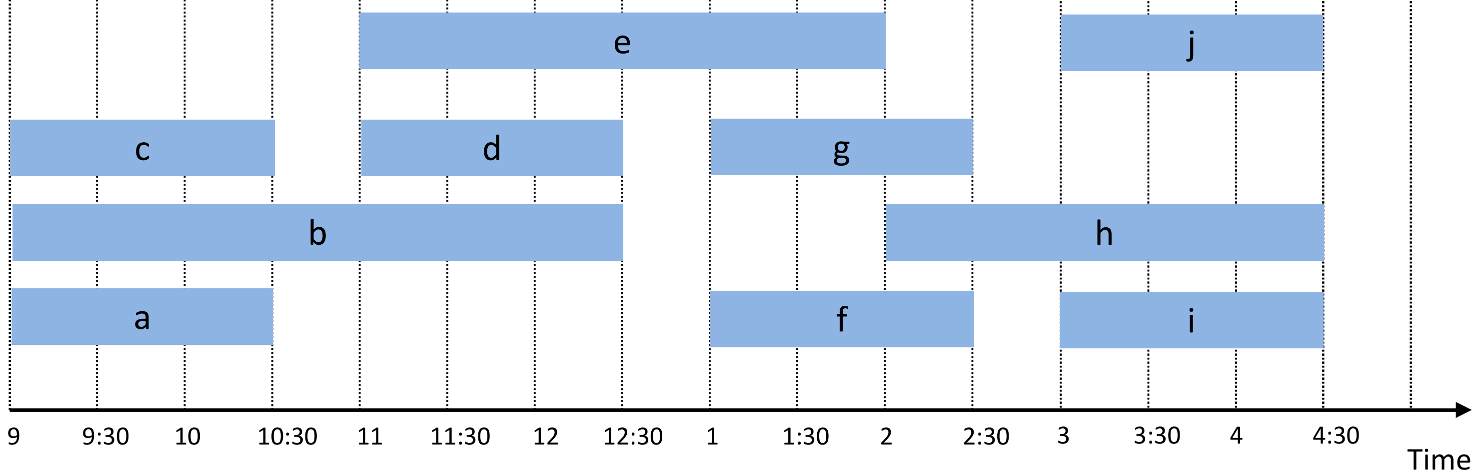
```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

A ← ϕ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

- Implementation. O(n log n).

Interval Partitioning

- Interval partitioning.
  - Lecture j starts at s_j and finishes at f_j.
  - Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

- Ex: This schedule uses 4 classrooms to schedule 10 lectures.



Interval Partitioning: Greedy Algorithm

- Greedy algorithm. Consider lectures in increasing order of start time: assign lecture to any compatible classroom (Don't open any new classroom unless necessary).

```
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤
sn.          ←—— number of allocated classrooms
d ← 0
for j = 1 to n {
   if (lecture j is compatible with some classroom k)
      schedule lecture j in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture j in classroom d + 1
      d ← d + 1
}
```

- Implementation. O(n log n).
  - For each classroom k, maintain the finish time of the last job added.
  - Keep the classrooms in a priority queue.

Definition of MST

Spanning tree: A subgraph T of a undirected and connected graph G=(V, E) is a spanning tree of G if

1. T is a tree and
2. T contains all the vertices (nodes) of G.

Growing a MST(Generic Algorithm)

```
GENERIC_MST(G, w)
    A:={}
    while A does not form a spanning tree do
        find an edge (u,v) that is safe for A
        A:=A∪{(u,v)}
    return A
```

- Set A is always a subset of some minimum spanning tree. This property is called the invariant property.
- An edge (u, v) is a safe edge for A if adding it to A does not destroy the invariant.

A is always part of a minimum spanning tree

Kruskal's algorithm

```
MST_KRUSKAL(G,w)
    A:={}
    for each vertex v in V[G]
        do MAKE_SET(v)
    sort the edges of E by nondecreasing weight w
    for each edge (u,v) in E, in order by nondecreasing
    weight
        do if FIND_SET(u) != FIND_SET(v)
                then  A:=A∪{(u,v)}
                      UNION(u,v)
    return A
```

Prim's algorithm

```
MST_PRIM(G, w, r)
    for each v in V do
    key[v]:=∞,  parent[v]:=NIL
    A:=Ø
key[r]:=0; parent[r]=NIL;
    Q←(V, key)  /* initialize Q.
    while Q!={} do
        u:=EXTRACT_MIN(Q); if parent[u]≠NIL, A:=A U (u, parent[u]).
        for each v in Adj[u] do
              if v in Q and w(u,v)<key[v]
              then    parent[v]:=u
                    k:=w(u,v)
                    Update(u,k)
```

Q (priority queue): contain all the vertices that have not yet been included in the tree. V\S: vertices in the tree

Parent[v]: the nearest vertex in the tree to v
Key[v]: the length of edge (v, parent[v])

i.e. u is not r.

u: the nearest vertex in Q to the tree.
Remove u from Q, i.e., add u to the tree

v has an edge with u

Single-Source Shortest Paths

- Problem Definition
- Shortest paths and Relaxation
- Dijkstra’s algorithm (a greedy algorithm)

Dijkstra's algorithm

$d[s] \leftarrow 0$

for each $v \in V - \{s\}$

  do $d[v] \leftarrow \infty$, $\pi[v] \leftarrow NIL$.

$S \leftarrow \varnothing$

$Q \leftarrow V$ % Q is a priority queue maintaining $V - S$

while $Q \neq \varnothing$

  do $u \leftarrow$ EXTRACT-MIN(Q)

    $S \leftarrow S \cup \{u\}$

relaxation step

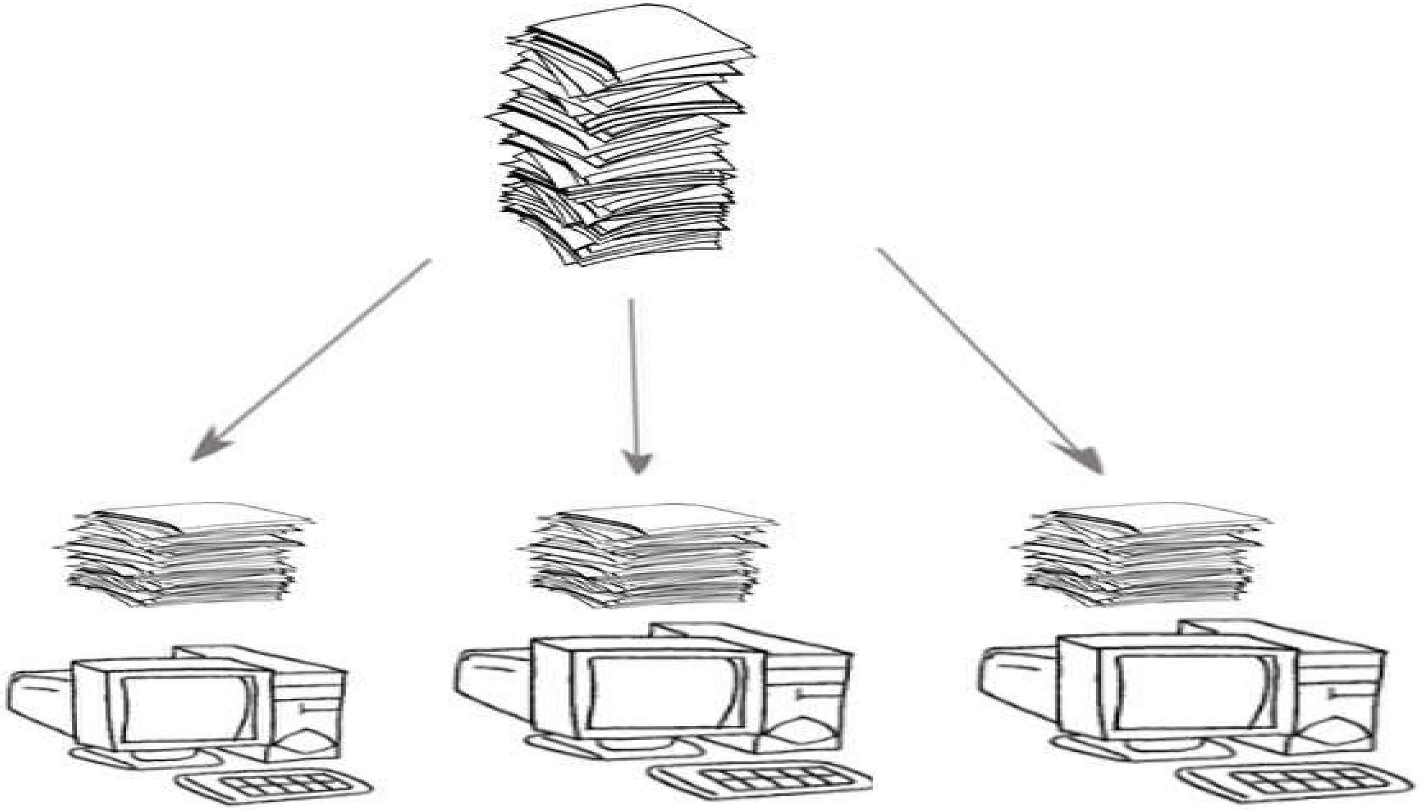
    for each $v \in Adj[u]$

      do if $d[v] > d[u] + w(u, v)$

        then $d[v] \leftarrow d[u] + w(u, v)$, $\pi[v] \leftarrow u$,

Implicit DECREASE-KEY

Divide and Conquer



Merge-Sort

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

Algorithm *mergeSort*(S)
    Input sequence S with n elements
    Output sequence S
        if $S.size() > 1$
        $(S_1, S_2) \leftarrow partition(S, n/2)$
        $mergeSort(S_1)$
        $mergeSort(S_2)$
        $S \leftarrow merge(S_1, S_2)$

A Useful Recurrence Relation

Def. T(n) = number of comparisons to mergesort an input of size n.

Mergesort recurrence.

$$T(n) \le \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Solution. $T(n) = O(n \log_2 n)$.

Assorted proofs. We describe several ways to prove this recurrence. Initially we assume n is a power of 2 and replace $\le$ with $=$.

Counting Inversions

Music site tries to match your song preferences with others.

- You rank n songs.
- Music site consults database to find people with similar tastes.

Similarity metric: number of inversions between two rankings.

- My rank: 1, 2, …, n.
- Your rank: $a_1, a_2, \ldots, a_n$.
- Songs i and j inverted if $i < j$, but $a_i > a_j$.

Songs

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

<u>Inversions</u>

3-2, 4-2

Brute force: check all $\Theta(n^2)$ pairs i and j.

Counting Inversions: Implementation

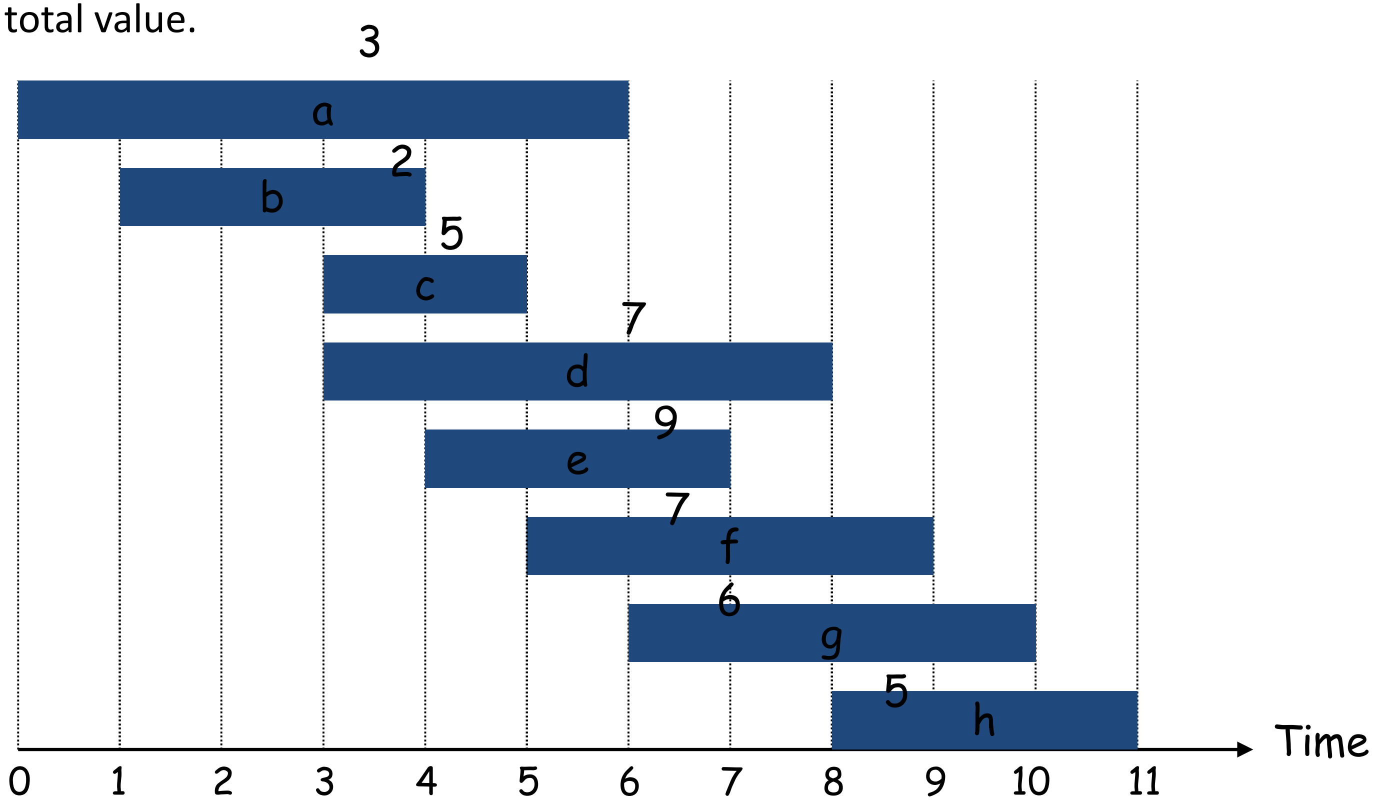
Pre-condition. [Merge-and-Count] A and B are sorted.
Post-condition. [Sort-and-Count] L is sorted.

```
Sort-and-Count(L) {
   if list L has one element
      return 0 and the list L

   Divide the list into two halves A and B
   (rA, A) ← Sort-and-Count(A)
   (rB, B) ← Sort-and-Count(B)
   (r , L) ← Merge-and-Count(A, B)

   return r = rA + rB + r and the sorted list L
}
```

Dynamic Programming

Weighted Interval Scheduling

- Weighted interval scheduling problem.
  - Job j starts at s_j, finishes at f_j, and has weight or value v_j .
  - Two jobs compatible if they don't overlap.
  - Goal: find a subset of pairwise compatible (nonoverlapping) jobs with maximal total value.



Dynamic Programming: Binary Choice

Notation. OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

- Case 1: OPT selects job j.
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

optimal substructure

- Case 2: OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

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

Weighted Interval Scheduling: Bottom-Up

Input: $n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n, v_1, v_2, \ldots, v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p(1), p(2), \ldots, p(n)$

M[0]=0; */* Memoization*

for j = 1 to n ***do***

M[j] = max $\{ v_j + M[p(j)], M[j-1] \}$

if *(M[j] == M[j-1])* ***then*** *B[j]=0* ***else*** *B[j]=1 /*for backtracking*

*m=n; /*** Backtracking*

B[j]=0 indicating job j is not selected.
B[j]=1 indicating job j is selected.

while *(m ≠0) {* ***if*** *(B[m]==1)* ***then***

print job m; m=p(m)

else

m=m-1 }

Time complexity

- **Sorting the jobs: O(n log n)**
- Computing p(): O(n) time after sorting all the jobs based on the starting times (O (n log n)) .
- The whole loop O(n) (each pass: O(1))
- The backtracking O(n)

•Time complexity: *O(n log n)* including sorting.

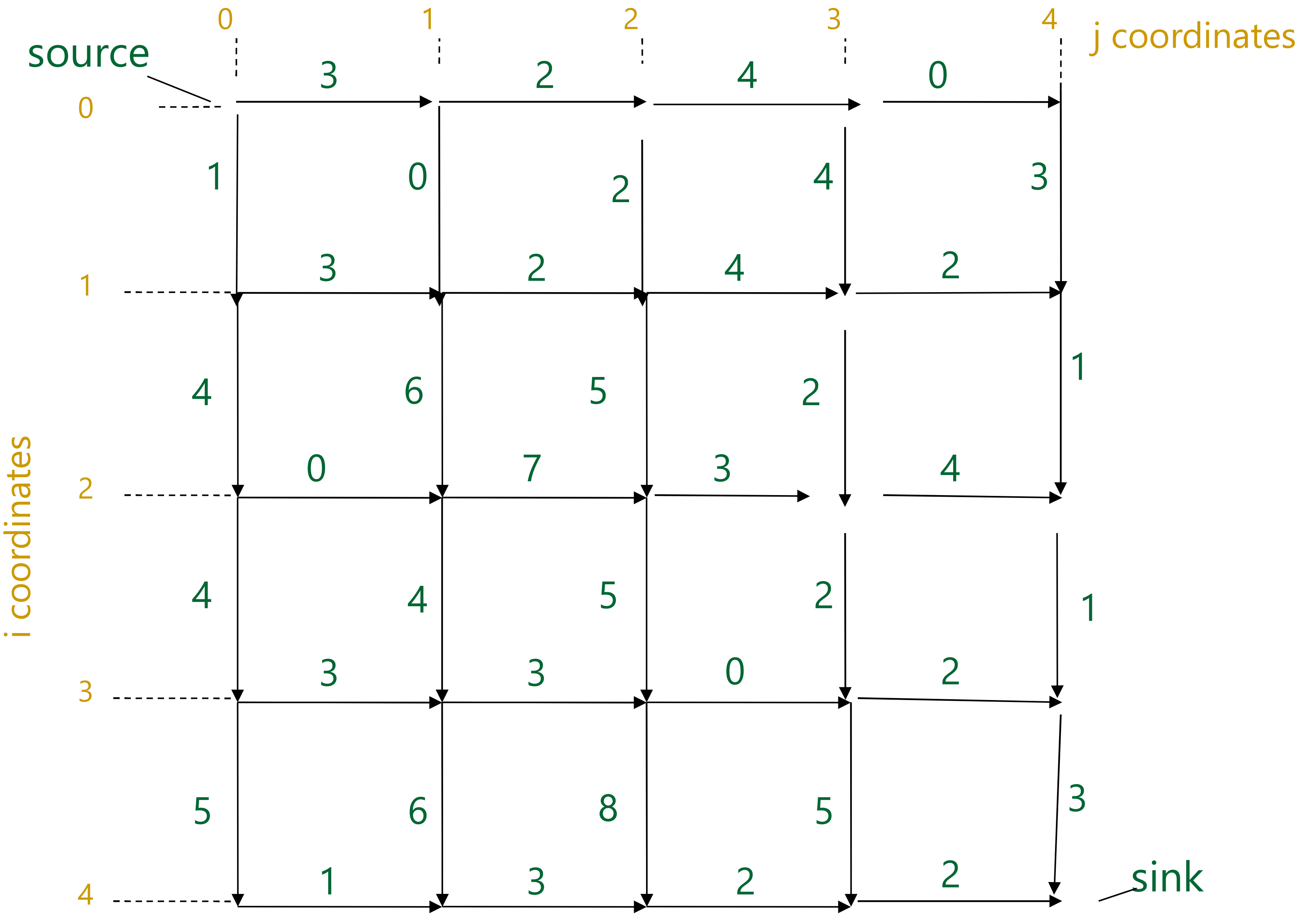
Manhattan Tourist Problem: Formulation

Goal: Find the longest path in a weighted grid

Input: A weighted grid **G** with two distinct vertices, one labeled “source” and the other labeled “sink”

Output: A longest path in **G** from “source” to “sink”

MTP: An Example



MTP: Simple Recursive Program

MT(n,m)

x ← MT(n-1,m)+length of the edge from (n- 1,m) to (n,m)

y ← MT(n,m-1)+length of the edge from (n,m-1) to (n,m)

return max{x,y}

MT(n,m): the length of the optimal path from (0,0) to (n,m)

0-1 Knapsack Problem

Knapsack Problem *0-1 version*

Knapsack problem.

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of W kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: { 3, 4 } has value 40.

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Greedy: repeatedly add item with maximum ratio v_i / w_i.
Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

Dynamic Programming: Adding a New Variable

Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w

- Case 2: OPT selects item i.
  - new weight limit = $w - w_i$
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT(i, w) = \begin{cases} 0 & \text{if } \mathrm{i} = 0 \\ OPT(i-1, w) & \text{if } \mathrm{w_i} > \mathrm{w} \\ \max\{\, OPT(i-1, w), \quad v_i + OPT(i-1, w-w_i) \,\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up

Knapsack. Fill up an n-by-W array.

```
Input: n, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

Knapsack Problem: Running Time

Running time. $\Theta(n\ W)$.

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete. [Chapter 8]

3. Longest common subsequence (LCS)

Longest common subsequence problem

- **Input:** Two sequences $X=x_1x_2\ldots x_m$, and $Y=y_1y_2\ldots y_n$.
- **Output:** a longest common subsequence of X and Y.

- **A brute-force approach**

Suppose that $m \geq n$. Try all subsequence of X (There are 2^m subsequence of X), test if such a subsequence is also a subsequence of Y, and select the one with the longest length.

The recursive equation

- Let c[i,j] be the length of an LCS of X[1...i] and Y[1...j].
- c[i,j] can be computed as follows:

$$c[i,j]=\begin{cases} 0 & \text{if } i=0 \text{ or } j=0, \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i=y_j, \\ \max\{c[i,j-1],c[i-1,j]\} & \text{if } i,j>0 \text{ and } x_i\neq y_j. \end{cases}$$

Computing the length of an LCS

- There are n×m c[i,j]'s. So we can compute them in
- a specific order.

The algorithm to compute an LCS

```
for i=1 to m do
        c[i,0]=0;
for j=0 to n do
        c[0,j]=0;
for i=1 to m  do
     for j=1 to n do
     {
           if x[i] ==y[j]  then
                c[i,j]=c[i-1,j-1]+1;
                b[i,j]=1;
           else   if c[i-1,j]>=c[i,j-1] then
                          c[i,j]=c[i-1,j]
                          b[i,j]=2;
                   else c[i,j]=c[i,j-1]
                          b[i,j]=3;
     }
```

4. Shortest common super-sequence

Recursive Equation:

- Let c[i,j] be the length of an SCS of X[1...i] and Y[1...j].
- c[i,j] can be computed as follows:

$$
c[i,j]=\begin{cases} j & \text{if } i=0 \\ i & \text{if } j=0, \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i=y_j, \\ \min\{c[i,j-1]+1, c[i-1,j]+1\} & \text{if } i,j>0 \text{ and } x_i\neq y_j. \end{cases}
$$

	y_i	B	D	**C**	A	**B**	**A**
x_i	0	1	2	3	4	5	6
A	1	← 2	← 3	← 4	↖ 4	← 5	↖ 6
B	2	↖ 2	← 3	← 4	← 5	↖ 5	← 6
C	3	↑ 3	← 4	↖ 4	← 5	← 6	← 7
B	4	↖ 4	← 5	↑ 5	← 6	↖ 6	← 7
D	5	↑ 5	↖ 5	← 6	← 7	↑ 7	← 8
A	6	↑ 6	↑ 6	← 7	↖ 7	← 8	↖ 8
B	7	↖ 7	↑ 7	← 8	↑ 8	↖ 8	↑ 9

The pseudo-codes

```
for i=0 to n do
   c[i, 0]=i;
for j=0 to m do
    c[0,j]=j;
for i=1 to n do
   for j=1 to m do
       if (xi == yj)  c[i ,j]= c[i-1, j-1]+1;  b[i.j]=1;
       else {
               c[i,j]=min{c[i-1,j]+1, c[i,j-1]+1}.
                if (c[i,j]=c[i-1,j]+1 then b[i,j]=2;
                else b[i,j]=3;
              }
 p=n, q=m;  / backtracking
 while (p≠0 or q≠0)
    { if (b[p,q]==1) then {print x[p]; p=p-1; q=q-1}
       if (b[p,q]==2) then {print x[p]; p=p-1}
       if (b[p,q]==3) then {print y[q]; q=q-1}
    }
```

Shortest Paths with Negative weighted edges

(***Bellman-Ford algorithm***)

Shortest Paths: Dynamic Programming

Def. OPT(i, v)=length of shortest s-v path P using at most i edges.

- Case 1: P uses at most i-1 edges.
  - OPT(i, v) = OPT(i-1, v)
- Case 2: P uses exactly i edges.
  - If (w, v) is the last edge, then OPT use the best s-w path using at most i-1 edges and edge (w, v).
  - $If\ i \geq 1, OPT(i,v) = \min\left\{OPT(i-1,v), \min_{(w,v)\in E}\{OPT(i-1,w) + c_{wv}\}\right\}$

  $Opt(0,s) = 0$

  If $i = 0$ *and* $v \neq s, OPT(i,v) = \infty$.

Remark: if no negative cycles, then OPT(n-1, v)=length of shortest s-v path.

Single-Source Shortest Paths in DAGs

- Shortest paths are always *well-defined* in *DAGs*
  - no cycles => no negative-weight cycles even if there are negative-weight edges

- **Idea:** If we were lucky
  - To process vertices on each shortest path from left to right, we would be done in 1 pass

Single-Source Shortest Paths in DAGs

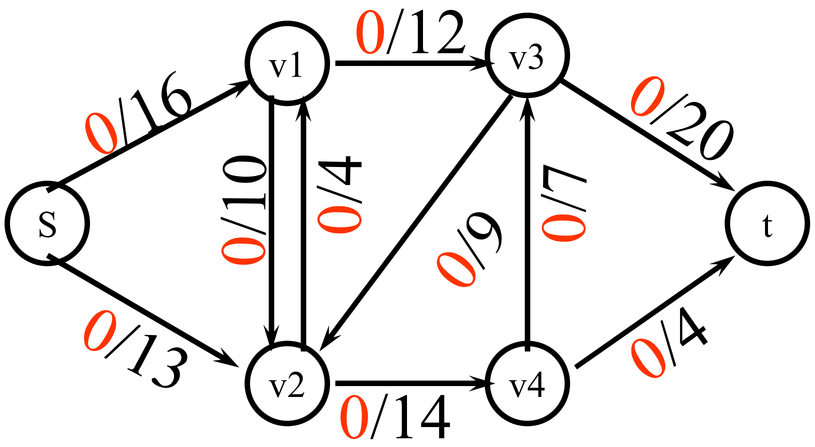
```
DAG-SHORTEST PATHS(G, s)
    TOPOLOGICALLY-SORT the vertices of G
    INIT(G, s)
    for each vertex u taken in topologically sorted order do
      for each v in Adj[u] do
          RELAX(u, v)
```

Maximum Flow

- Maximum Flow Problem
- The Ford-Fulkerson method
- Maximum bipartite matching

The basic Ford Fulkerson algorithm

example of an execution



for *each edge (u, v) ∈ E [G]*
  do f [u, v] = 0
    f [v, u] = 0
while *there exists a path p from s to t in the residual network* G_f
  do $c_f(p) = \min\{c_f(u, v) \mid (u, v) \in p\}$
    for *each edge (u, v) in p*
      do $f[u, v] = f[u, v] + c_f(p)$
        $f[v, u] = - f[u, v]$

The Edmonds-Karp algorithm

- Find the augment path using breadth-first search.
- Breadth-first search gives the shortest path for graphs (Assuming the length of each edge is 1.)
- Time complexity of Edmonds-Karp algorithm is $O(V*E^2)$.
- The proof is very hard and is not required here.

Class P and Class NP

- Class P contains problems which are solvable in polynomial time.
  - –The problems have algorithms in $O(n^k)$ time, where n is the input size and k is a constant.
- Class NP (nondeterministic polynomial) consists of those problems that are *verifiable* in polynomial time.
  - we can verify that the solution is correct in time polynomial in the input size to the problem.

Example: Hamilton Circuit: given an order of the n distinct vertices $(v_1, v_2, \ldots, v_n)$, we can test if (v_i, v_{i+1}) is an edge in G for $i=1, 2, \ldots, n-1$ and (v_n, v_1) is an edge in G in time $O(n)$ (polynomial in the input size).

Some basic NP-complete problems

- **3-Satisfiability** : Each clause contains at most three variables or their negations.
- **Vertex Cover**: Given a graph G=(V, E), find a subset V' of V such that for each edge (u, v) in E, at least one of u and v is in V' and the size of V' is minimized.
- **Hamilton Circuit**: (definition was given before)
- History: Satisfiability→3-Satisfiability→vertex cover→Hamilton circuit.
- Those proofs are very hard.

Approximation Algorithm

An algorithm *A* is an approximation algorithm

- if given any instance *I,* it finds a candidate solution *s(I)*

How good an approximation algorithm is?

We use *performance ratio* to measure the quality of an approximation algorithm.

Performance ratio

For minimization problem, the performance ratio r of algorithm A is defined as for any instance I of the problem,

$$\boxed{\frac{A(I)}{OPT(I)} \le r \qquad (r \ge 1)}$$

where OPT(I) is the value of the optimal solution for instance I and A(I) is the value of the solution returned by algorithm A on instance I.