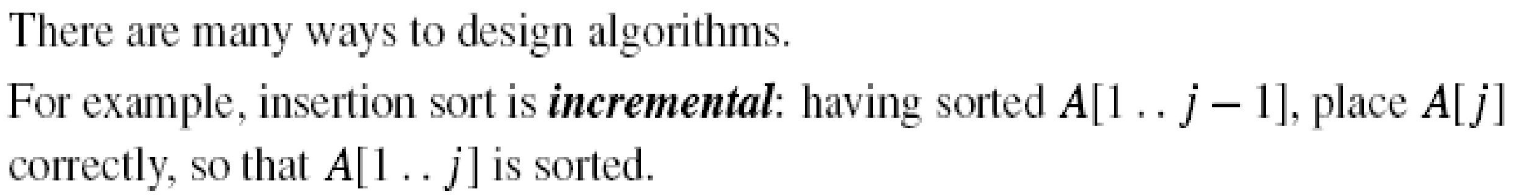
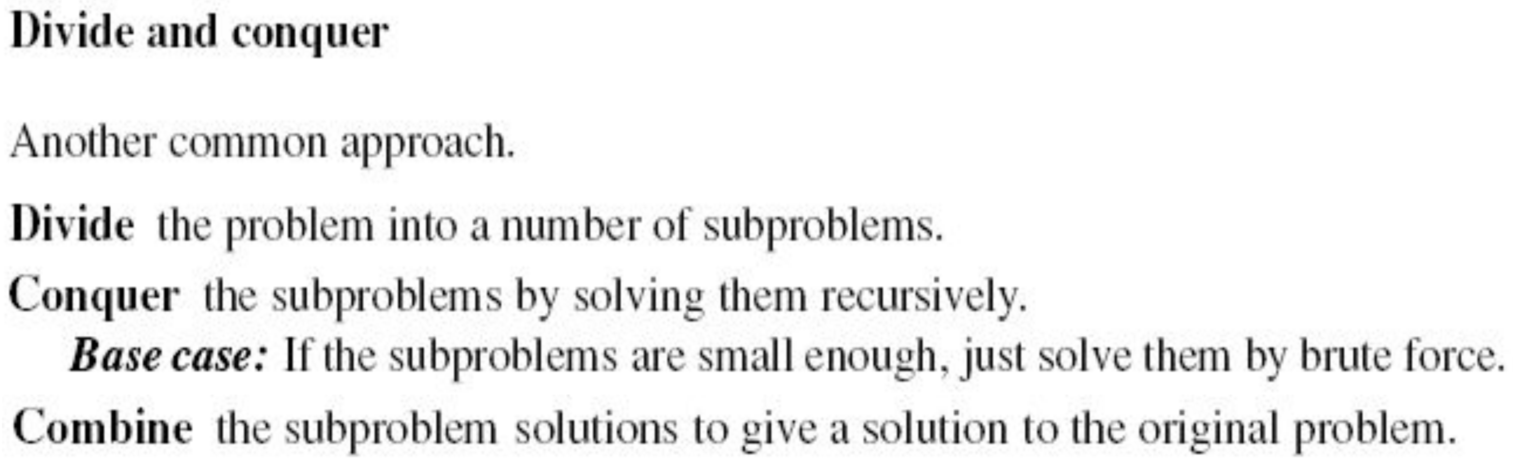
Divide and conquer algorithms

General template


There are many ways to design algorithms.

For example, insertion sort is ***incremental***: having sorted $A[1 \mathinner{..} j-1]$, place $A[j]$ correctly, so that $A[1 \mathinner{..} j]$ is sorted.


Divide and conquer

Another common approach.

Divide the problem into a number of subproblems.

Conquer the subproblems by solving them recursively.

  Base case: If the subproblems are small enough, just solve them by brute force.

Combine the subproblem solutions to give a solution to the original problem.

To sort $A[p..r]$:

Divide by splitting into two subarrays $A[p..q]$ and $A[q+1..r]$, where q is the halfway point of $A[p..r]$.

Conquer by recursively sorting the two subarrays $A[p..q]$ and $A[q+1..r]$.

Combine by merging the two sorted subarrays $A[p..q]$ and $A[q+1..r]$ to produce a single sorted subarray $A[p..r]$. To accomplish this step, we'll define a procedure MERGE(A, p, q, r).

The recursion bottoms out when the subarray has just 1 element, so that it's trivially sorted.

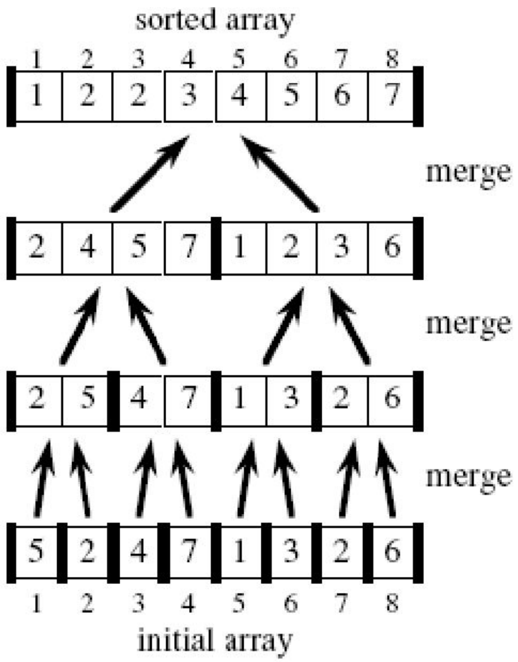
Merge sort

```
MERGE-SORT($A, p, r$)
if $p < r$                              ▷ Check for base case
  then $q \leftarrow \lfloor (p + r)/2 \rfloor$              ▷ Divide
       MERGE-SORT($A, p, q$)            ▷ Conquer
       MERGE-SORT($A, q + 1, r$)        ▷ Conquer
       MERGE($A, p, q, r$)              ▷ Combine
```

Initial call: MERGE-SORT($A, 1, n$)

```
MERGE($A, p, q, r$)
$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
create arrays $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$
for $i \leftarrow 1$ to $n_1$
    do $L[i] \leftarrow A[p + i - 1]$
for $j \leftarrow 1$ to $n_2$
    do $R[j] \leftarrow A[q + j]$
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow p$ to $r$
    do if $L[i] \leq R[j]$
          then $A[k] \leftarrow L[i]$
               $i \leftarrow i + 1$
          else $A[k] \leftarrow R[j]$
               $j \leftarrow j + 1$
```

Example: Bottom-up view for $n = 8$: *[Heavy lines demarcate subarrays used in subproblems.]*



Analyzing divide-and-conquer algorithms

Use a ***recurrence equation*** (more commonly, a ***recurrence***) to describe the running time of a divide-and-conquer algorithm.

Let $T(n)$ = running time on a problem of size n.

- If the problem size is small enough (say, $n \le c$ for some constant c), we have a base case. The brute-force solution takes constant time: $\Theta(1)$.
- Otherwise, suppose that we divide into a subproblems, each $1/b$ the size of the original. (In merge sort, $a = b = 2$.)
- Let the time to divide a size-n problem be $D(n)$.
- There are a subproblems to solve, each of size $n/b \Rightarrow$ each subproblem takes $T(n/b)$ time to solve $\Rightarrow$ we spend $aT(n/b)$ time solving subproblems.
- Let the time to combine solutions be $C(n)$.
- We get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c\ , \\ aT(n/b) + D(n) + C(n) & \text{otherwise}\ . \end{cases}$$

Analyzing merge sort

For simplicity, assume that n is a power of 2 $\Rightarrow$ each divide step yields two subproblems, both of size exactly $n/2$.

The base case occurs when $n = 1$.

When $n \geq 2$, time for merge sort steps:

Divide: Just compute q as the average of p and $r \Rightarrow D(n) = \Theta(1)$.

Conquer: Recursively solve 2 subproblems, each of size $n/2 \Rightarrow 2T(n/2)$.

Combine: MERGE on an n-element subarray takes $\Theta(n)$ time $\Rightarrow C(n) = \Theta(n)$.

Since $D(n) = \Theta(1)$ and $C(n) = \Theta(n)$, summed together they give a function that is linear in n: $\Theta(n) \Rightarrow$ recurrence for merge sort running time is

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

Solving the merge-sort recurrence: By the master theorem we can show that this recurrence has the solution $T(n) = \Theta(n \lg n)$. *[Reminder:* $\lg n$ *stands for* $\log_2 n$.*]*

```
BinarySearch(A[0..n-1], value, low, high)
{
while (low <= high)
{
   mid = (low + high) / 2
   if (A[mid] > value)
    return BinarySearch(A, value, low, mid-1)
   else if (A[mid] < value)
    return BinarySearch(A, value, mid+1, high)
   else
    return mid              // found
 }
return  -1           // not found
}
```

Greedy Algorithms

Introduction

Similar to dynamic programming.

Used for optimization problems.

Idea: When we have a choice to make, make the one that looks best *right now*. Make a *locally optimal choice* in hope of getting a *globally optimal solution*.

Greedy algorithms don't always yield an optimal solution. But sometimes they do. We'll see a problem for which they do. Then we'll look at some general characteristics of when greedy algorithms give optimal solutions.

Problem

- A town has a set of houses and a set of roads.
- A road connects 2 and only 2 houses.
- A road connecting houses u and v has a repair cost $w(u, v)$.
- ***Goal:*** Repair enough (and no more) roads such that

  1. everyone stays connected: can reach every house from all other houses, and
  2. total repair cost is minimum.

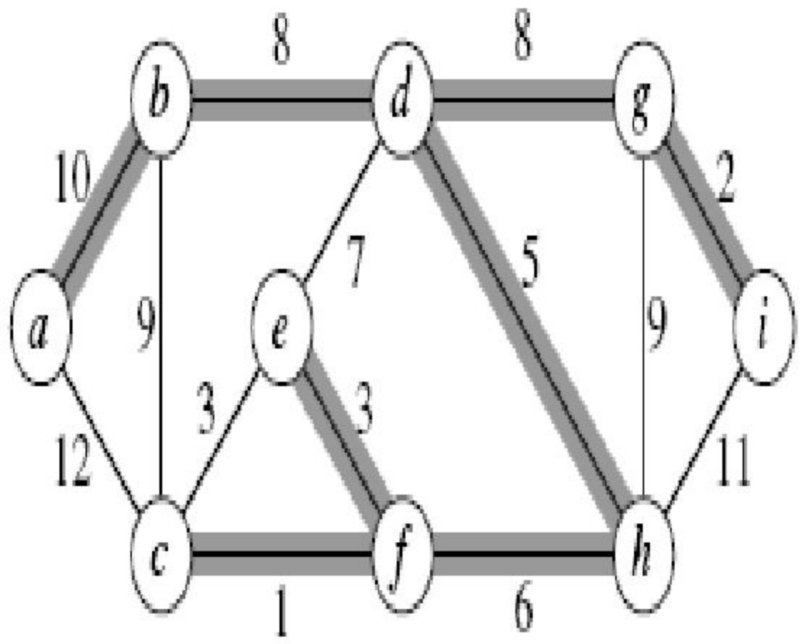
Model as a graph:

- Undirected graph $G = (V, E)$.
- ***Weight*** $w(u, v)$ on each edge $(u, v) \in E$.
- Find $T \subseteq E$ such that

  1. T connects all vertices (T is a ***spanning tree***), and
  2. $w(T) = \sum_{(u,v) \in T} w(u, v)$ is minimized.

A spanning tree whose weight is minimum over all spanning trees is called a ***minimum spanning tree***, or ***MST***.

Example of such a graph *[edges in MST are shaded]* :



In this example, there is more than one MST. Replace edge (e, f) by (c, e). Get a different spanning tree with the same weight.

Kruskal's algorithm

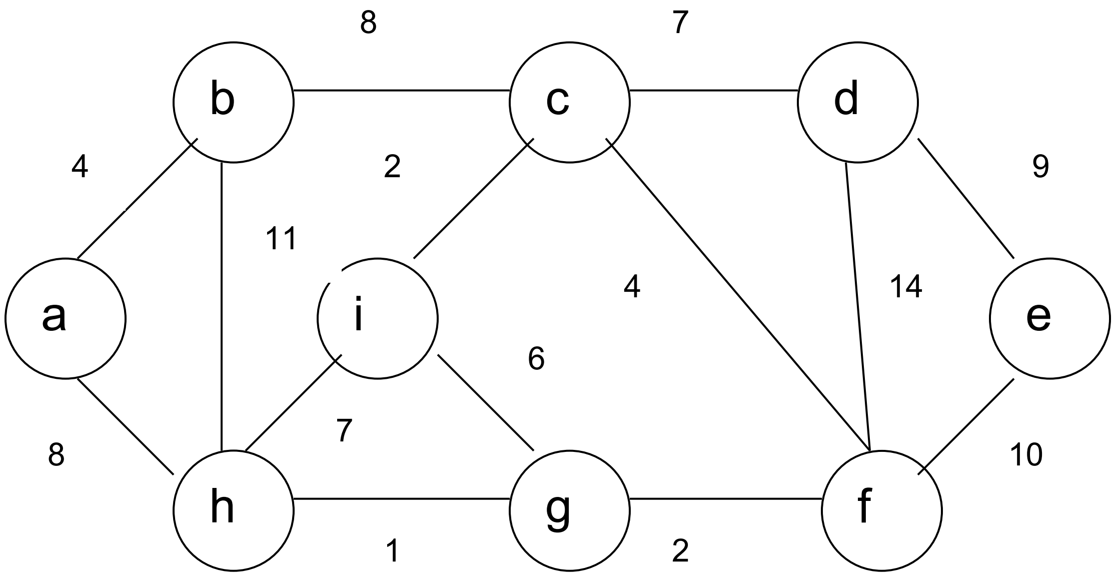
```
KRUSKAL($V$, $E$, $w$)
$A \leftarrow \emptyset$
for each vertex $v \in V$
    do MAKE-SET($v$)
sort $E$ into nondecreasing order by weight $w$
for each $(u, v)$ taken from the sorted list
    do if FIND-SET($u$) $\neq$ FIND-SET($v$)
          then $A \leftarrow A \cup \{(u, v)\}$
               UNION($u$, $v$)
return $A$
```

Run through the above example to see how Kruskal's algorithm works on it:

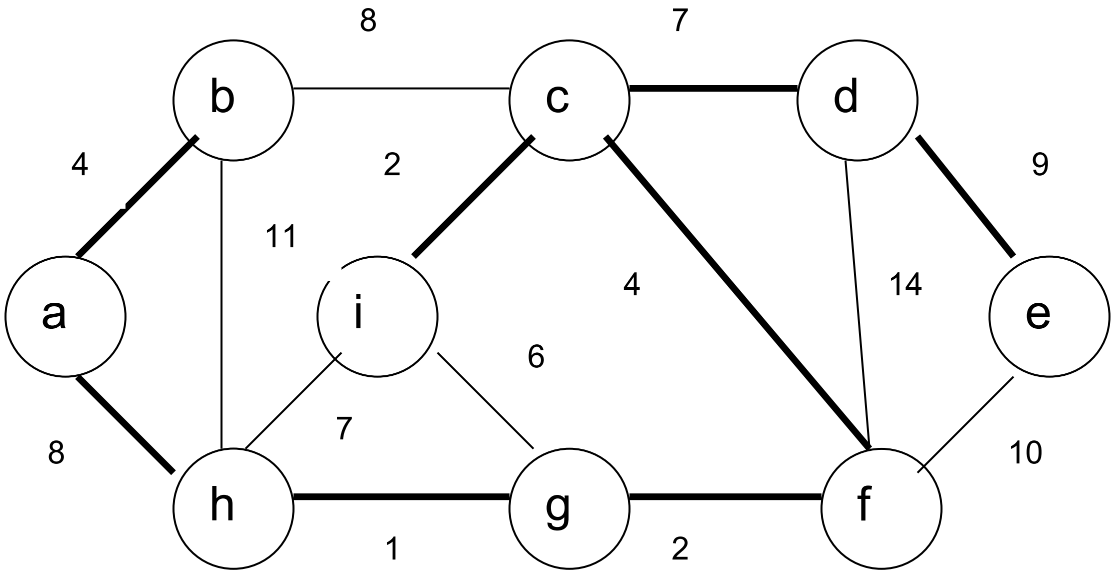
(c, f)	:	safe
(g, i)	:	safe
(e, f)	:	safe
(c, e)	:	reject
(d, h)	:	safe
(f, h)	:	safe
(e, d)	:	reject
(b, d)	:	safe
(d, g)	:	safe
(b, c)	:	reject
(g, h)	:	reject
(a, b)	:	safe

At this point, we have only one component, so all other edges will be rejected. *[We could add a test to the main loop of* KRUSKAL *to stop once $|V| - 1$ edges have been added to A.]*

Ex.



Solution:



Analysis

Initialize A:	$O(1)$
First for loop:	$\lvert V \rvert$ MAKE-SETs
Sort E:	$O(E \lg E)$
Second for loop:	$O(E)$ FIND-SETs and UNIONs

- Assuming the implementation of disjoint-set data structure,

  $O((V + E)\,\alpha(V)) + O(E \lg E)$.

- Since G is connected, $|E| \geq |V| - 1 \Rightarrow O(E\,\alpha(V)) + O(E \lg E)$.
- $\alpha(|V|) = O(\lg V) = O(\lg E)$.
- Therefore, total time is $O(E \lg E)$.
- $|E| \leq |V|^2 \Rightarrow \lg |E| = O(2 \lg V) = O(\lg V)$.

- Therefore, $O(E \lg V)$ time. (If edges are already sorted, $O(E\,\alpha(V))$, which is almost linear.)

Dijkstra's algorithm

No negative-weight *edges*.

Essentially a weighted version of breadth-first search.

- Instead of a FIFO queue, uses a priority queue.
- Keys are shortest-path weights ($d[v]$).

Have two sets of vertices:

- S = vertices whose final shortest-path weights are determined,
- Q = priority queue = $V - S$.

```
DIJKSTRA($V, E, w, s$)
INIT-SINGLE-SOURCE($V, s$)
$S \leftarrow \emptyset$
$Q \leftarrow V$        ▷ i.e., insert all vertices into $Q$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN($Q$)
       $S \leftarrow S \cup \{u\}$
       for each vertex $v \in Adj[u]$
           do RELAX($u, v, w$)
```

- $\pi[v]$ = predecessor of v on a shortest path from s.

All the shortest-paths algorithms start with INIT-SINGLE-SOURCE.

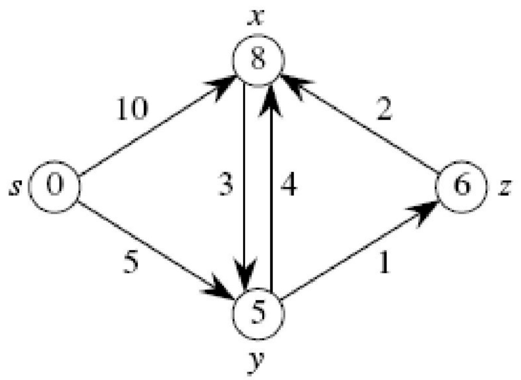
```
INIT-SINGLE-SOURCE($V$, $s$)
for each $v \in V$
    do $d[v] \leftarrow \infty$
       $\pi[v] \leftarrow$ NIL
$d[s] \leftarrow 0$
```

Relaxing an edge (u, v)

Can we improve the shortest-path estimate for v by going through u and taking (u, v)?

```
RELAX($u$, $v$, $w$)
if $d[v] > d[u] + w(u, v)$
  then $d[v] \leftarrow d[u] + w(u, v)$
       $\pi[v] \leftarrow u$
```

Example:



Order of adding to S: s, y, z, x.

Analysis: If priority queue is implemented as binary heap : -

- Each Extract-Min operations then takes time O(lg V)
- There are |V| such operations
- The time to build the binary min-heap is O(V)
- Each Decrease-Key operation (implicit in Relax) takes time O(lg V), and there are at most |E| such operations
- Total running time: O((V+E) lg V)
  => O(E lg V)

Because each vertex v is added to set S exactly once, each edge in the adjacency list Adj[v] is examined in the for loop exactly once during the course of the algorithm. Since the total number of edges in all the adjacency lists is |E|, there are a total of |E| iterations of this for loop. { Note: We are using aggregate analysis }

Scheduling

- Theorem: The greedy method always obtains an optimal solution to the job sequencing problem.

Algorithm for job scheduling using greedy strategy

{ Note: We want to find a feasible solution S whose profit P(S) is as large as possible.
}

```
Sort the jobs in decreasing order of profits: $g_1$>=…>=$g_n$
d □ max $d_i$              // $d_i$ stands for deadline for job i
for t: 1..d        // t stands for time slot
  S(t) □ 0               // S stands for schedule
end for
 for i: 1..n
Find the largest t such that (S(t)=0 and t<=$d_i$), and let S(t) □ i
[Find the latest possible free slot meeting the deadline]
end for
```

{ Note: If S(t) = 0, then no job is scheduled at time t, g_i is a non-negative real no. representing the profit obtainable from job i. If S(t)=i, then job i is scheduled at time t, 1<=t<=d.}

Ex. Let n=4, (P1,P2,P3,P4)=(100,10,15,27) and (d1,d2,d3,d4)=(2,1,2,1)
Solution:
Decreasing order of profits: 100 27 15 10
d=2

0	0

27	100

Ex.: Let n=5, (P1,..,P5)=(20,15,10,5,1) and (d1,…,d5)=(2,2,1,3,3). Find the optimal solution.
Solution: The optimal solution is :

15	20	5

Ex. Given below is a problem of scheduling unit-time tasks with deadlines and penalties for a single processor.

Tasks

i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Solution:

40	60	50	70		10

Total penalty incurred is $w_5 + w_6 = 50$