Algorithms

Basic DSes

Array vs LL

Arrays: Static data sets, ones where fast index-based access is needed, Slow expansion, insertion, deletion. **LLs:** Dynamic data sets, insertion & deletion more important than random access.

Linked Lists

Operations

Search	O(N)
Prepend / Append	O(1)
Delete	O(N)

Runtime Analysis

constant	1
logarithmic	$\log N$
linear	N
linearithmic	$N \log N$
quadratic	N^2
cubic	N^3
exponential	2^N

Sorting Algorithm Info

Algorithm	$O(N)$	$\Omega(N)$	$\Theta(N)$
Selection	$O(N^2)$	$\Omega(N^2)$	$\Theta(N^2)$
Insertion	$O(N^2)$	$\Omega(N)$	$\Theta(N^2)$
Shellsort	$\leq O(N^2)$	$\Omega(N \log N)$	$\Theta(N \log N)$
Mergesort	$O(N \lg N)$	$\Omega(N \lg N)$	$\Theta(N \lg N)$
Quicksort	$O(N^2)$	$\Omega(N \lg N)$	$\Theta(N \lg N)$
Heapsort	$O(N \log N)$	$\Omega(N \log N)$	$\Theta(N \log N)$

Algorithm	Stbl?	In-Place?	Space
Selection Sort	N	Y	O(1)
*Insertion Sort	Y	Y	O(1)
Shellsort	N	Y	O(1)
**Quicksort	N	Y	O(log N)
Mergesort	Y	N	O(N)
Heapsort	N	Y	O(1)

* depends on order of items
** probabilistic guarantee

Tilde Approximation

$$\lim_{n\to\infty} \frac{\sim f(n)}{f(n)} = 1$$

TL;DR drop everything but the most significant factor in terms of n.
Order of growth – Just drop the constant from the tilde approximation

Time Complexity

Notation Definitions

- $O(g(n))$ – upper limit
- $\Omega(g(n))$ – lower limit
- $\Theta(g(n))$ – upper and lower limit

```
for(i=0;i<n;i++)     - O(n)
for(i=0;i<n;i=i+2)   - n/2 O(n)
for(i=n;i>1;i--)     - O(n)
for(i=1;i<n;i=i*2)   - O(log_2 n)
for(i=1;i<n;i=i*3)   - O(log_3 n)
for(i=n;i>1;i=i/2)   - O(log_2 n)
```

Let $c_1, c_2 > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$,

$$0 \leq f(n) \leq c_1 g(n) \quad \Rightarrow f(n) \in O(g(n))$$
$$c_1 g(n) \leq f(n) \quad \Rightarrow f(n) \in \Omega(g(n))$$
$$c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \Rightarrow f(n) \in \Theta(g(n))$$

Limit Definitions

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty \quad \Rightarrow f(n) \in O(g(n))$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0 \quad \Rightarrow f(n) \in \Omega(g(n))$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \notin \{0, \infty\} \quad \Rightarrow f(n) \in \Theta(g(n))$$

Memory Complexity

In-place algorithms: $N + O(\ln N)$
Mergesort:
- Original input array: N.
- Aux array for merging: $O(N)$.
- Local variables: *Const*.
- Function call stack: $O(\log N)$.
- Total: $O(2N \log N)$.

Union-Find

Mutually connected nodes are called components.

API

- **Connected(n_id1, n_id2)**: returns true if n_id1 and n_id2 are connected.
- **Union(n_id1, n_id2)**: if not connected, connects two nodes
- **Find(n_id)**: takes node id, returns union n_id is in
- **Count(n_id)**: return the size of n_id's union

Time Complexity

Algo	Q-F	Q-U	WQ-U
Initialization	$O(N)$	$O(N)$	$O(N)$
Find	$O(1)$	$O(N)$	$O(lg(N))$
Connected	$O(1)$	$O(N)$	$O(lg(N))$
Union	$O(N)$	$O(N)$	$O(lg(N))$

Quick-Find

- Associate nodes with unions
- represented with an array
- Indices are node ids
- Values are union ids (we choose one node id for the union id)

Initialization

Initialize an length N array arr with arr[i] = i.

Union(n_id1, n_id2)

Change every occurance of n_id1 to n_id2 in arr.

Find(n_id)

returns arr[n_id].

Connected

returns if arr[n_id1] == arr[n_id2].

Quick-Union

- Associate nodes with other nodes
- nodes point to their parent node, which lead to the root node/union id
- root nodes point to themselves

Initialization

Initialize a length N array arr with arr[i] = i.

Union

Takes indices p and q, finds the root nodes. If they are not equal, make one point to the other.

Find

Takes an index i, finds arr[i] until the root node. Returns the root node id.

Connected

Takes an indices p and q, finds the root nodes. Returns true if they are equal.
Given a class with an int array *id*

```
QUICK-UNION(N)
  id = INTEGERARRAY(N)
  for i = 0 to N
      id[i] = i

FIND(i)
  while not id[i] == i
      i = id[i]
  return i

UNION(p, q)
  i = FIND(p)
  j = FIND(q)
  id[i] = j

CONNECTED(p, q)
  return FIND(p) == FIND(q)
```

Weighted Quick-Union

To optimize Quick-Union, we try to create the flattest trees possible. We need another array to count the number of nodes rooted at an index. Union now links the root of the smaller tree to the root of the larger tree. This can be further optimized with path compression, which can be tacked on to find. To do this, we change each node to point at it's grandparent.

Given a class with an int array *id* and an int array *sz*

```
UNION(p, q)
  i = FIND(p)
  j = FIND(q)
  id[i] = j
  if sz[i] < sz[j]
      id[i] = j
      sz[j] = sz[i]
  else id[j] = i
      sz[i] = sz[j]
```

Bad Sorts

Selection Sort

Build up a sorted section of array, by

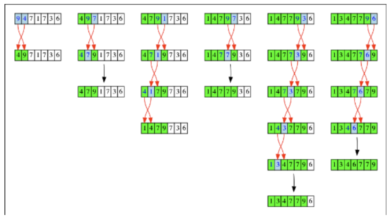
1. Find min element
2. Swap min & first element
3. Examine array, skipping first element.

Pros: Minimal number of write operations (eg tiny RAM?) **Cons:** Slow as shit $\Theta(N^2)$

Insertion Sort

- Start from 2 elements, build up sorted subarray, "inserting" a new element each iteration by swapping until it is in the right position.
- # of operations depends on the degree of disorder (how unsorted the array is).

Pros: If the array has a low degree of disorder, it is faster. **Cons:** Worst case is still slow as shit.

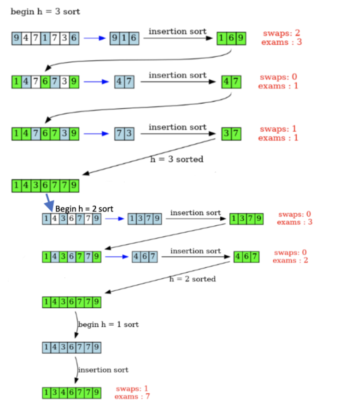


$\Omega(N), O(n^2)$

Shellsort

- Pick every h elements and put them in a subarray.
- Sort the subarrays using insertion sort.
- Repeat until $h = 1$.

Pros: Used when: in embedded systems applications from using small program size and memory efficiency. Reduces large amounts of disorder quickly. **Cons:** Slow



```
public class Shell {
  public static void sort(
        ↪ Comparable[] a) {
    int N = a.length;
    int h = 1;
    while (h < N/3) h = 3 * h + 1;
    while (h >= 1) {
      for (int i = h; i < N; i++) {
        for (int j = i; j >= h &&
            ↪ less(a[j], a[j-h]);
            ↪  j -= h)
          exch(a, j, j-h);
      }
      h = h / 3;
    }
  }
}
```

Good Sorts

Mergesort

Divide array in half recursively, until it is down to 1 element. Merge array together like a zipper.
Time Complexity: $O(n \log n)$
Memory Complexity: $O(N)$

Code

```
private static void merge(
     ↪ Comparable[] a, Comparable
     ↪ [] aux, int lo, int mid,
     ↪ int hi) {
  // copy
  for (int k = lo; k <= hi; k++)
    aux[k] = a[k];
  // merge
  int i = lo, j = mid + 1;
  for (int i = lo; k <= hi; k++) {
    if (i > mid)
      a[k] = aux[j++];
    else if
      (j > hi) a[k] = aux[i++];
    else if
      (less(aux[j], aux[i])) a[k] =
         ↪  aux[j++];
    else
      a[k] = aux(i++);
  }
}

private static void sort(Comparable
     ↪ [] a, Comparable[] aux, int
     ↪  lo, int hi) {
  if (hi <= lo) return;
  int mid = lo + (hi - lo) / 2;
  sort(a, aux, lo, mid);
  sort(a, aux, mid+1, hi);
  merge(a, aux, lo, mid, hi);
}

public static void sort(Comparable
     ↪ [] a) {
  Comparable[] aux = new Comparable
       ↪ [a.length];
  sort(a, aux, 0, a.length - 1);
}
```

Top-down vs Bottom-up TL;DR

Top-down uses recursion: starts at **top** of tree and proceeds **downwards**. Bottom-up does not use recursion: starts at **bottom** of tree and iterates over pieces moving **upwards**.

Bottom-up

Pass through array, merging as we go to double size of sorted subarrays. Keep performing the passes and merging subarrays, until you do a merge that encompasses the whole array.

```
public class MergeBU {
  private static Comparable[] aux;
  // see above for merge() code
  public static void sort(
        ↪ Comparable[] a) {
    int N = a.length;
    aux = new Comparable[N];
    for (int sz = 1; sz < N; sz =
          ↪ sz + sz) // sz:
          ↪ subarray size
      for (int lo = 0; lo < N - sz;
            ↪  lo += sz + sz) // lo
            ↪ : subarray index
        merge(a, lo, lo+sz-1, Math.
              ↪ min(lo+sz+sz-1, N
              ↪ -1));
  }
}
```

Min Top-down Comparisons
Proposition: Top-down mergesort uses between $1/2NlogN$ and $NlogN$ comparisons.
Let the total number of comparisons be C(N): $C(\lceil N/2 \rceil)$ = left recursion, $C(\lfloor N/2 \rfloor)$ = right recursion, cN = comparisons at this level, $C(N) \leq C(\lceil N/2 \rceil) + C(\lfloor N/2 \rfloor) + cN$. The smallest number of comparisons made by *MERGE* is $N/2$. If one sub-array contains all the smallest elements, we only walk that array before appending the other. If you sort a sorted array, this would be the case at every level. Solving the above recursion with $c = 1/2$ yields $C(N) = 1/2NlogN$
Max top-down Comparisons:
Similarly, the maximum number of comparisons is made when both sub-arrays must be fully examined. If the input array is of size N , then at most N comparisons are made. If the above happens at every level of the recursion, the total number of comparisons at each level is at most N. By solving the same recurrance equation with $c = 1$, we get $C(N) = NlogN$
Max top-down Accesses:
Proposition. Top-down mergesort uses at most $6NlogN$ array accesses to sort an array of length N.
Each merge uses at most 6N array accesses $2N$ to copy the sub arrays initially $2N$ to put the values back (in order) At most N comparisons, each accessing two array elements ($2N$)Hence the total number of array accesses after solving the recurrence is most $6NlogN$.
Min/Max Bottom Up Comparisons/Accesses: *Proposition. Bottom-up mergesort uses between $1/2NlogN$ and $NlogN$ compares and at most $6NlogN$ array accesses to sort an array of length N.* The number of passes through the array is precisely $logN$. For each pass, by the same argument made as in the case of the top-down mergesort approach, the number of array accesses is exactly $6N$ and the number of compares is at most N and no less than $N/2$. Therefore, bottom-up mergesort uses between $1/2NlogN$ and $NlogN$ compares and at most $6NlogN$ array accesses to sort an array of length N.

Quicksort

1. Shuffle array to reduce impact of order on sorting speed
2. Pick first element of array as pivot
3. Create two sub arrays from remaining elements, one selecting those smaller, one selecting those larger. Put them on either side of the pivot
4. Recurse for each side of the pivot until everything is sorted.

```
partition (arr, lo, hi):
  pivot = hi
  i = lo
  for j from lo+1 to hi:
    if arr[j]<pivot:
      swap(arr[i], arr[j])
      i++

quicksort (arr, lo, hi) {
  if lo<hi:
    pivot = partition(arr, lo,
       ↪ hi)
    quicksort(arr, lo, pivot−1)
    quicksort(arr, pivot+1, hi)
```

- Fastest for disordered arrays, slowest for already sorted arrays
- Randomize array or select a random piviot to prevent worst case. (Best choise of a piviot is the median)
- **Best case**: The partitions are always of equal size : $\Omega(NlogN)$. Recurrence relation is $T(n) = 2T(n/2) + cn$.
- **Worst case**: One partition is always of size 0 (if the array is already sorted and we are picking pivots from the ends) : $O(N^2)$. Recurrence relation is $T(n) = T(n-1) + T(0) + cn$.
- **Average case**: 1.39 $\log N \in \Theta(NlogN)$
- Uses less memory than merge sort. Space complexity $O(n)$

Priority Queues

Supports insertion and removing/popping the priority (largest or smallest) item. Implementations:

Sorted Array - O(n) insert, O(1) pop

Unsorted Array - O(1) insert, O(n) pop

Binary Heap - O(log n) insert and sort

Binary Heaps & Heapsort

A max binary heap is a complete binary tree where the keys are in the nodes and each parent's key $\geq$ each child's key. This requirement is called the max heap property. Binary Heaps:

- Can be represented as array or tree/nodes.
- Insertion: We insert at the end of the array, then "swim up" the value.
- Swimming up - exchange a given node with it's parent until the binary max property is fulfilled.
- Popping - swap the first node (the max) with the last node, remove it, then "sink" the first node.
- Sinking - exchange a given node with the max of it's children until the binary max property is fulfilled.

Heapsort relies on the binary heap data structure to sort data. Once a max heap has been constructed, you can perform a single for loop and call $RemoveMax$ to build a sorted array (thus linearizing the heap).

MaxPQ

isEmpty

return n == 0

insert(Key x)

insert x at pq[n], increment n, then swim(n)

delMax

set max to pq[1], decrement n, exch(1, n), sink(1), set pq[n+1] to null, return max

swim(int k)

while k > 1 and pq[k/2] < pq[k]: exch(k, k/2), k = k/2

sink(int k)

while (2k $\leq$ n): Exchange the parent with the larger child (children of k are at 2k and 2k+1)

Binary Trees

Binary Tree: A tree where each internal node has at most two children. **Full Binary Tree:** A binary tree where each internal node has exactly two children. **Complete Binary Tree:** A complete binary tree is which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
Internal Node: Any node that has children $L = N - 1$, where L is the number of the internal node in the tree, and N is the number of leaves.
$N = 2^h$, where h is the height of the tree and N is the number of nodes in the tree. To go the other way around, $\log_2(N) = h$.
Maximum height: $n - 1$ **Minimum height:** floor(lg n)

Symbol Tables

Operations: Insert a new pair into the table (set) Search for the value associated with a given key (get) Equality test required, inequality operator allows for an ordered symbol table. Allows for new operations and faster runtimes at the cost of key monotyping.
Implementations

imp	(wrst) srch	ins	del	(avg) srch	ins	del	ord ops?
seq	N	N	N	$.5N$	N	$.5N$	n
bin	lg N	lg N	lg N	lg N	$.5N$	$.5N$	y
BST	N	N	N	1.39 lg N	1.39 lg N	$\sqrt{N}$	y
23	c lg N	c lg N	c lg N	c lg N	c lg N	c lg N	y
RB BST	2 lg N	2 lg N	2 lg N	1.0 lg N	1.0 lg N	1.0 lg N	y

Sequential Search (Unordered LL) Search: $O(N)$, Insert: $O(N)$ Pros: Best for tiny STs Cons: Slow for large STs
Binary Search

```
BINARY-SEARCH($A$, $x$)
  $a = 0$ // Lower Bound
  $b = A.length - 1$ // Upper Bound
  while $a \leq b$
      $m = \text{floor}(a + (b - a)/2)$
      if $x < A[m]$
          $b = m - 1$
      elseif $x > A[m]$
          $a = m + 1$
      else return $m$
  return NIL
```

$O(\log N)$ time complexity, and at most $1 + \log_2 N$ compares. Search: $O(\lg N)$, Insert: $O(N)$ Pros: Optimal search and space, order-based operations Cons: Slow insert

Binary Search Trees

Definitions: Depth of Node: length of path from root to node (root has depth 0) Height of Node: length of longest path from node to leaf (leaf has height 0) Height of Tree: length of longest path from root to leaf (empty tree has height -1) Degree of Node: number of subtrees attached to a node (degree of tree is max of degrees of nodes) Key, value, pointers to left and right subtrees, N for node count in subtree (left contains strictly smaller elements, right contains strictly larger elements)
Find: traverse tree, comparing desired element to current node's key (trivial) `get()` does the same thing except we return the value / NULL `insert()`: traverse tree, inserting node where we would expect to find it `put()`: Overwrite value if key already exists, otherwise add new node. Comparisons is 1 + depth of node
`min()`: go all the way left `max()`: go all the way right
`floor()`: largest key less than or equal to key
`ceiling()`: smallest key greater than or equal to key
Lazy Deletion: Set value to null, leave key in tree to guide search
Hibbard Deletion:

- If node to be deleted has no children, delete it
- If node to be deleted has one child, replace with child
- If node to be deleted has both children, replace with minimum key in right subtree

Traversals

1. Inorder (Left, Root, Right)
2. Preorder (Root, Left, Right)
3. Postorder (Left, Right, Root)

Balanced Search Trees

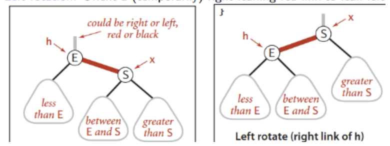
2-3 Trees

2-nodes have one key and two children 3-nodes have two keys and three children
Insert: if leaf is a 2-node, make it into a 3-node If leaf is a 3-node, make a 4 node, and decompose it into two 2-nodes, passing the middle element up to the parent (if exists)
Tree height: Worst case: $\log_2 N$ Best case: $\log_3 N \approx .631 log_2 N$
See above for time complexity

LLRB Trees

Just a way to encode 2-3 trees more simply (they exactly correspond). Black links are the same in both, red links connect values that are 3-nodes in 2-3 trees.
Rules: No node has two red links connected to it. Every path from root to null link has the same number of black links - (Perfect Black Balance) Red links lean left.

Left rotation. Orient a (temporarily) right-leaning red link to lean left



Color flip: Turn a 4-node into a 2-node by flipping the colors of the child links and the parent link
Height is $\leq 2\log_2 N$ in the worst case See above for time complexity

Hash Tables

TLDR replace index access with hash function, use either a probing strategy or chaining to resolve collisions

Good Hash

Division method $h(k) = k \mod m$ hashes k into one of m slots. A prime number not too close to 2^p is a good choice.
Multiplication method
$h(k) = \lfloor m(kA \mod 1) \rfloor$ Value of m is not critical, generally picked to be a power of 2 to make implementation easy. A is a constant between 0 and 1, and is usually chosen to be $\frac{\sqrt{5}-1}{2}$.

Chaining

Put all elements with the same hash value into the same linked list. For linked list T, worst case time complexity is $O(1)$ for insert, $O(m)$ for search and delete where $m = |T|$.

Linear Probing

$h(k, i) = (h'(k) + i) \mod m$ Where $h'(k)$ is a hash function, and i is the probe number. Increment i until an empty slot is found. Suffers from primary clustering (long runs of occupied slots tend to build up)

Quadratic Probing

$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m$ Where $h'(k)$ is a hash function, c_1 and c_2 are constants, and i is the probe number. Doesn't encounter primary clustering, but two keys with the same initial probe position leads to the same probe sequence (secondary clustering).

Double Hashing

$h(k, i) = (h_1(k) + ih_2(k)) \mod m$ Where $h_1(k)$ and $h_2(k)$ are hash functions, and i is the probe number. Doesn't encounter either form of clustering. h_2 should be relative prime to m. One way to do this is to let m be prime, and pick h_2 such that it always returns a positive integer less than m. Eg: $h_1(k) = k \mod m$, $h_2(k) = 1 + (k \mod (m'))$, where m' is a prime less than m (eg $m - 1$)

Undirected Graphs

If an edge exists between two vertices, they are adjacent, and the edge is incident to both verticies. Degree of vertex is number of edges incident to it. Two edges that connect the same pair of vertices are parallel.
Adjacency-matrix: 2D V by V boolean array
Adjacency-list: Vertex-indexed array of lists, each list element is adjacent to index. Can be implemented with a list of Bags, allowing for parallel and self loops.

DFS

Simple, used to find all vertices connected to a source, or to find a path between two vertices.

```
DFS($G$, $v$)
  $marked[v]$ = TRUE
  for $i$ = 0 to ADJ($G$, $v$).$length$
      $w$ = ADJ($G$, $v$)[$i$]
      if $marked[w]$ == FALSE
          $edgeTo[w]$ = $v$
          DFS($G$, $w$)
```

Time Complexity $\rightarrow O(|V| + |E|)$

BFS

Used to find the shortest path between two vertices.

```
BFS($G$, $s$)
  $queue$ = NEWQUEUE()
  $marked[s]$ = TRUE
  ENQUEUE($queue$, $s$)
  while ISEMPTY($queue$) == FALSE
      $v$ = DEQUEUE($queue$)
      for $i$ = 0 to ADJ($G$, $v$).$length$
          $w$ = ADJ($G$, $v$)[$i$]
          if $marked[w]$ == FALSE
              $edgeTo[w]$ = $v$
              $marked[w]$ = TRUE
              ENQUEUE($queue$, $w$)
```

Time Complexity $\rightarrow O(|V| + |E|)$

Connected components

Init all vertices v as unmarked, for each unmarked vertex v, run DFS to find all vertic1es connected to v. Space and time proportional to $(V + E)$: each adjacency list entry is examined only once, and there are 2E such entries (two for each edge). If preprocessing is viable, it is faster than union-find. However, union-find is online, and as such can be used when doing few queries, or when data is not already structured as a graph.

Directed Graphs

Terminology

Indegree $\rightarrow$ number of edges pointing to a vertex
Outdegree $\rightarrow$ number of edges pointing away from a vertex

API

same as undir graphs, except we have $reverse()$ which produces a new graph with all edges reversed.

Topological Sort

same as DFS, except when you hit a node that has no adjacent nodes, add that node to a list post[]. After DFS is over, reverse the list, that's your topological order

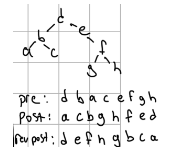
Multiple-source shortest paths:

Given digraph and set of source vertices, find shortest path from and V $\in$ set to each other vertexes.

- Do BFS but initialize by enqueing all source vertices to queue
- keeps track using edgeTo[] to find shortest path among nodes

Reverse Postorder:

Do DFS, but add vertices to a stack as soon as you mark them as visited. When DFS is over, pop the stack to get the reverse postorder.



Strongly connected components:

Two vertices are strongly connected if there is a directed path from v to w and from w to v. In a DAG, we can have atmost V strongly connected components.

Kosaraju algo (Find strong components in a dir graph)

- Run DFS on G^r and compute reverse post order
- Run DFS on G, considering verticies in ordere given by the reverse post order

MSTs

Edge-weighted graph: Undirected graph model where weights/costs are associated with each Edge
Kruskals and Prim's used to find MSTs (minimum spanning trees) Difference: Prim's uses a PQ, Kruskal's uses a greedy approach

Kruskal's

1. use a min priority queue P to store edges $O(|E|)$ for init
2. pop the min edge from P. $O(|E| \log |E|)$ in total
3. while examining an edge, check for cycle. $O(|E| + |V|)$ in total
4. if e does not create a cycle, then add e to T

Total Time complexity of $O(|E| \log |E|)$ checking for cycles: use union find. if v and w are in the same component, then adding $v - w$ creates a cycle

Prim's

Start with vertex 0 and greedily grow tree T
Consider edges incident on vertices in T, but disregard edges with both end points in T
Add T to min weighted edge with exactly one endpoint in T
Repeat until $V - 1$ edges.
Lazy: Maintain a PQ of edges with at least one endpoint in T
Key = edge; priority = weight of edge
Delete-min edge $e = v - w$ from PQ to find next edge to add to T
Disregard if both endpoints are marked (in T)
Otherwise, let w be the unmarked vertex:
- add to PQ any edge incident to w (assuming other endpoint not in T)
- add e to T and mark w

operation	frequency	binary heap
delete min	E	log E
insert	E	log E

Eager: Maintain a PQ of verticies connected by an edge to T, where priority of vertex v = *min. weighted edge* connecting v to T
Delete min vertex v, mark v to be in T
Update PQ by considering all edges $e = v - x$ incident to v - ignore if x is already in T
- add x to PQ if not already there

- if already on PQ, then reduce priority of x if $v - x$ becomes the min. weighted edge connecting x to T

Uses extra space proportional to $|V|$ and time proportional to $|E| \log |V|$ (worst case) for E edges and V vertices.