CS 580: Algorithm Design and Analysis

Jeremiah Blocki
Purdue University
Spring 2019

Reminder: Homework 1 due tonight at 11:59PM! (Gradescope)

Recap: Greedy Algorithms

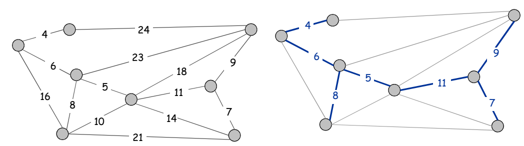
Minimize Lateness
- **Greedy Choice:** Sort by earliest deadline
- **Proof of Optimality:** can always optimal solution into one with fewer inversions (Greedy Choice has 0 inversions)
- **Running Time:** O(n log n)

Optimal Offline Caching
- **Goal:** Minimize number of cache misses
- **Greedy Choice:** Evict item used furthest in future [Belady'60]
- **Proof of Optimality:** Invariant: S_{FF} is optimal through first j+1 requests.
- **Limitation:** Need to know sequence in advance.

2

4.5 Minimum Spanning Tree

Minimum Spanning Tree



G = (V, E)   T, $\Sigma_{e \in T}\, c_e = 50$

Minimum spanning tree. Given a connected graph G = (V, E) with real-valued edge weights c_e, an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree whose sum of edge weights is minimized.

Cayley's Theorem. There are n^{n-2} spanning trees of K_n.

↑ can't solve by brute force

4

Applications

MST is fundamental problem with diverse applications.

- Network design.
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems.
  - traveling salesperson problem, Steiner tree
- Indirect applications.
  - max bottleneck paths
  - LDPC codes for error correction
  - image registration with Renyi entropy
  - learning salient features for real-time face verification
  - reducing data storage in sequencing amino acids in a protein
  - model locality of particle interactions in turbulent fluid flows
  - autoconfig protocol for Ethernet bridging to avoid cycles in a network
- Cluster analysis.

5

Greedy Algorithms

Kruskal's algorithm. Start with T = ϕ. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.

Reverse-Delete algorithm. Start with T = E. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T.

Prim's algorithm. Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T.

Remark. All three algorithms produce an MST.

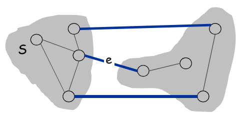
6

Greedy Algorithms

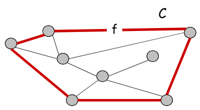
Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e.

Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f.



e is in the MST

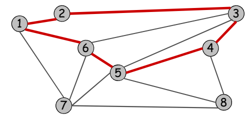


f is not in the MST

7

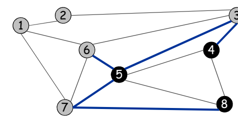
Cycles and Cuts

Cycle. Set of edges of the form a-b, b-c, c-d, …, y-z, z-a.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1

Cutset. A cut is a subset of nodes S. The corresponding cutset D is the subset of edges with exactly one endpoint in S.

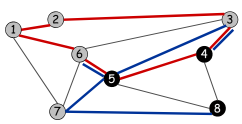


Cut S = { 4, 5, 8 }
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8

8

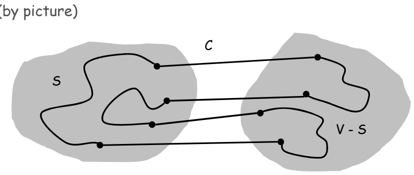
Cycle-Cut Intersection

Claim. A cycle and a cutset intersect in an even number of edges.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersection = 3-4, 5-6

Pf. (by picture)



9

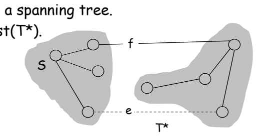
Greedy Algorithms

Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST T* contains e.

Pf. (exchange argument)

- Suppose e does not belong to T*, and let's see what happens.
- Adding e to T* creates a cycle C in T*.
- Edge e is both in the cycle C and in the cutset D corresponding to S $\Rightarrow$ there exists another edge, say f, that is in both C and D (even #edges in intersection).
- T' = T* $\cup$ { e } - { f } is also a spanning tree.
- Since $c_e < c_f$, cost(T') < cost(T*).
- This is a contradiction. ▪



10

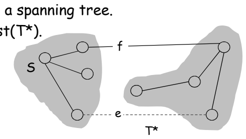
Greedy Algorithms

Simplifying assumption. All edge costs c_e are distinct.

Cycle property. Let C be any cycle in G, and let f be the max cost edge belonging to C. Then the MST T* does not contain f.

Pf. (exchange argument)

- Suppose f belongs to T*, and let's see what happens.
- Deleting f from T* creates a cut S in T*.
- Edge f is both in the cycle C and in the cutset D corresponding to S $\Rightarrow$ there exists another edge, say e, that is in both C and D.
- T' = T* $\cup$ { e } - { f } is also a spanning tree.
- Since $c_e < c_f$, cost(T') < cost(T*).
- This is a contradiction. ▪

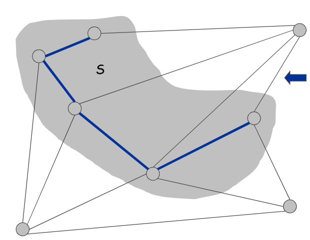


11

Prim's Algorithm: Proof of Correctness

Prim's algorithm. [Jarník 1930, Dijkstra 1959, Prim 1957]

- Initialize S = any node.
- Apply cut property to S.
- Add min cost edge in cutset corresponding to S to tree T, and add one new explored node u to S.



12

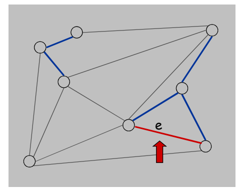
Implementation: Prim's Algorithm

Implementation. Use a priority queue ala Dijkstra.
- Maintain set of explored nodes S.
- For each unexplored node v, maintain attachment cost a[v] = cost of cheapest edge v to a node in S.
- $O(n^2)$ with an array; $O(m \log n)$ with a binary heap;
- $O(m + n \log n)$ with Fibonacci Heap

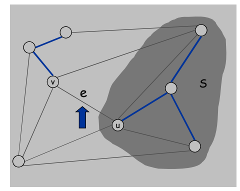
```
Prim(G, c) {
   foreach (v ∈ V) a[v] ← ∞
   Initialize an empty priority queue Q
   foreach (v ∈ V) insert v onto Q
   Initialize set of explored nodes S ← φ

   while (Q is not empty) {
      u ← delete min element from Q
      S ← S ∪ { u }
      foreach (edge e = (u, v) incident to u)
         if ((v ∉ S) and (c_e < a[v]))
            decrease priority a[v] to c_e
}
```


Kruskal's Algorithm: Proof of Correctness

Kruskal's algorithm. [Kruskal, 1956]
- Consider edges in ascending order of weight.
- Case 1: If adding e to T creates a cycle, discard e according to cycle property.
- Case 2: Otherwise, insert e = (u, v) into T according to cut property where S = set of nodes in u's connected component.



Case 1



Case 2

Implementation: Kruskal's Algorithm

Implementation. Use the union-find data structure.
- Build set T of edges in the MST.
- Maintain set for each connected component.
- $O(m \log n)$ for sorting and $O(m\, \underbrace{\alpha(m, n)}_{\text{essentially a constant}})$ for union-find.

$m \le n^2 \Rightarrow \log m$ is $O(\log n)$

```
Kruskal(G, c) {
   Sort edges weights so that c_1 ≤ c_2 ≤ ... ≤ c_m.
   T ← φ

   foreach (u ∈ V) make a set containing singleton u

   for i = 1 to m              are u and v in different connected components?
      (u,v) = e_i
      if (u and v are in different sets) {
         T ← T ∪ {e_i}
         merge the sets containing u and v
      }                        merge two components
   return T
}
```


Lexicographic Tiebreaking

To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts to break any ties.

Impact. Kruskal and Prim only interact with costs via pairwise comparisons. If perturbations are sufficiently small, MST with perturbed costs is MST with original costs.

e.g., if all edge costs are integers, perturbing cost of edge e_i by i / n^2

Implementation. Can handle arbitrarily small perturbations implicitly by breaking ties lexicographically, according to index.

```
boolean less(i, j) {
   if      (cost(e_i) < cost(e_j)) return true
   else if (cost(e_i) > cost(e_j)) return false
   else if (i < j)                 return true
   else                            return false
}
```


MST Algorithms: Theory

Deterministic comparison based algorithms.
- $O(m \log n)$ [Jarník, Prim, Dijkstra, Kruskal, Boruvka]
- $O(m \log \log n)$. [Cheriton-Tarjan 1976, Yao 1975]
- $O(m\, \beta(m, n))$. [Fredman-Tarjan 1987]
- $O(m \log \beta(m, n))$. [Gabow-Galil-Spencer-Tarjan 1986]
- $O(m\, \alpha(m, n))$. [Chazelle 2000]

Holy grail. $O(m)$.

Notable.
- $O(m)$ randomized. [Karger-Klein-Tarjan 1995]
- $O(m)$ verification. [Dixon-Rauch-Tarjan 1992]

Euclidean.
- 2-d: $O(n \log n)$. compute MST of edges in Delaunay
- k-d: $O(k n^2)$. dense Prim

4.7 Clustering

Clustering

Clustering. Given a set U of n objects labeled $p_1, \ldots, p_n$, classify into coherent groups.

photos, documents. micro-organisms

Distance function. Numeric value specifying "closeness" of two objects.

number of corresponding pixels whose intensities differ by some threshold

Fundamental problem. Divide into clusters so that points in different clusters are far apart.
- Routing in mobile ad hoc networks.
- Identify patterns in gene expression.
- Document categorization for web search.
- Similarity searching in medical image databases
- Skycat: cluster 10^9 sky objects into stars, quasars, galaxies.

19

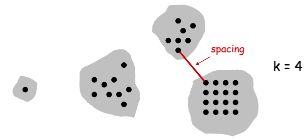
Clustering of Maximum Spacing

k-clustering. Divide objects into k non-empty groups.

Distance function. Assume it satisfies several natural properties.
- $d(p_i, p_j) = 0$ iff $p_i = p_j$ (identity of indiscernibles)
- $d(p_i, p_j) \geq 0$ (nonnegativity)
- $d(p_i, p_j) = d(p_j, p_i)$ (symmetry)

Spacing. Min distance between any pair of points in different clusters.

Clustering of maximum spacing. Given an integer k, find a k-clustering of maximum spacing.



20

Greedy Clustering Algorithm

Single-link k-clustering algorithm.
- Form a graph on the vertex set U, corresponding to n clusters.
- Find the closest pair of objects such that each object is in a different cluster, and add an edge between them.
- Repeat n-k times until there are exactly k clusters.

Key observation. This procedure is precisely Kruskal's algorithm (except we stop when there are k connected components).

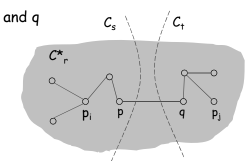
Remark. Equivalent to finding an MST and deleting the k-1 most expensive edges.

21

Greedy Clustering Algorithm: Analysis

Theorem. Let C^* denote the clustering $C^*_1, \ldots, C^*_k$ formed by deleting the k-1 most expensive edges of a MST. C^* is a k-clustering of max spacing.

Pf. Let C denote some other clustering $C_1, \ldots, C_k$.
- The spacing of C^* is the length d^* of the $(k-1)^{st}$ most expensive edge (in MST).
- Let p_i, p_j be in the same cluster in C^*, say C^*_r, but different clusters in C, say C_s and C_t.
- Some edge (p, q) on p_i-p_j path in C^*_r spans two different clusters in C.
- All edges on p_i-p_j path have length $\leq d^*$ since Kruskal chose them.
- Spacing of C is $\leq d^*$ since p and q are in different clusters. ▪



22

Union Find Data-Structure

Union-Find Operations

Three Operations

- MakeUnionFind(S)
  - Initialize a Union-Find data structure where all elements in S are in separate sets
- Find(u)
  - **Input:** $u \in S$
  - **Output:** Name of the set A containing u
  - **Require:** If u,v in the same set A then Find(u)=Find(v)
- Union(A,B)
  - **Input:** Names of sets A and B in the Union-Find data structure
  - **No Output:** Merge the sets A and B into a single set $A \cup B$
  - **Require:** If we had $u \in A$ and $v \in B$ then we require that Find(u)=Find(v) after this operation is completed

24

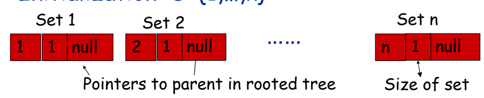
Union-Find Applications

- Efficient Implementation of Kruskal's Algorithm
- Initially all nodes are in different sets (no edges added to T)
  - Find(u) = u for each node u
  - Indicates that each node is its own connected component (initially)
- Add edge (u,v) to T
  - Merges two connected components containing u and v respectively
  - Union(A,B) where A = Find(u) and B=Find(v)
- Check if adding edge (u,v) induces a cycle in T
  - **Observation:** (u,v) induces a cycle if and only u and v are already in the same connected component.
  - **Test:** Find(u) = Find(v)?
    - Yes → u,v are in same component (u,v) would induce cycle
    - No → u,v are not in same component (u,v) won't induce cycle

25

Union-Find Implementation

MakeUnionFind(S)
  Initialization: S={1,…,n}



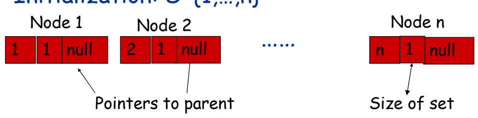

```
List<Node> Sets;

MakeUnionFind(n)
   for (i=1 to n) {
      Node v;
      v.Index = i;
      v.Size = 1;
      v.Parent = null;
      Sets.Add(v)
   }
}
```

```
struct Node {
   int Index;
   Node * Parent;
   int Size;
}
```

26

Union-Find Implementation

MakeUnionFind(S)
  Initialization: S={1,…,n}

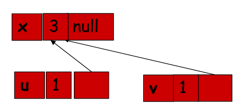



```
node Find(v) {
   if (v.parent == null)
       return v
   else
       vRoot =Find(v.parent)
       return vRoot
}
```

27

```
node Find(v) {
   if (v.parent == null)
       return v
   else
       vRoot =Find(v.parent)
       return vRoot
}
```

Example: Find(v) = x



28

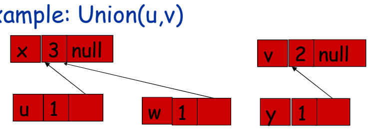
Union-Find Implementation

```
Union(Node u, Node v){
  uRoot = Find(u),  vRoot=Find(v)
  if (uRoot=vRoot) return
  else if (uRoot.size > vRoot.size)
      vRoot.Parent = uRoot; uRoot.size+= vRoot.size;
  else
      uRoot.Parent = vRoot; vRoot.size+= uRoot.size;
}
```

uRoot is new root of Merged set

vRoot is new root of Merged set

Example: Union(u,v)

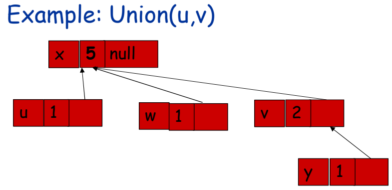


29

Union-Find Implementation

```
Union(Node u, Node v){
  uRoot = Find(u),  vRoot=Find(v)
  if (uRoot=vRoot) return
  else if (uRoot.size > vRoot.size)
      vRoot.Parent = uRoot; uRoot.size+= vRoot.size;
  else
      uRoot.Parent = vRoot; vRoot.size+= uRoot.size;
}
```

Example: Union(u,v)

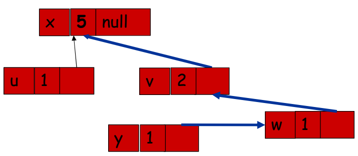


30

Path Compression

```
node Find(v) {
   if (v.parent == null)
       return v
   else
       vRoot =Find(v.parent)
       v.Parent = vRoot;
       return vRoot
}
```

Example: Find(y)

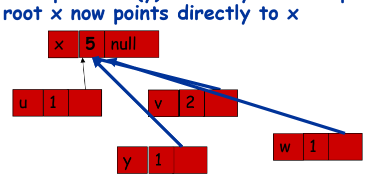


31

Path Compression

```
node Find(v) {
   if (v.parent == null)
       return v
   else
       vRoot =Find(v.parent)
       v.Parent = vRoot;
       return vRoot
}
```

Example: Find(y) - every node on path from y to root x now points directly to x



32

Union Find: Running Time

(Path Compression + Union by Size)

- Amortized Running Time: $O(\alpha(n))$ per operation
- $\alpha(n)$ - Inverse Ackermann Function
- (Grows Incredibly Slowly)
- $\alpha(n) \leq 5$ for any value of n you will ever use on a computer!
- Could achieve same result with union by rank (height of tree)

33

MST Algorithms: Theory

Deterministic comparison based algorithms.

- O(m log n) [Jarník, Prim, Dijkstra, Kruskal, Boruvka]
- O(m log log n). [Cheriton-Tarjan 1976, Yao 1975]
- O(m β(m, n)). [Fredman-Tarjan 1987]
- O(m log β(m, n)). [Gabow-Galil-Spencer-Tarjan 1986]
- O(m α (m, n)). [Chazelle 2000]

Holy grail. O(m).

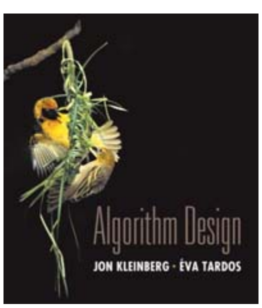
Notable.

- O(m) randomized. [Karger-Klein-Tarjan 1995]
- O(m) verification. [Dixon-Rauch-Tarjan 1992]

Euclidean.

- 2-d: O(n log n). compute MST of edges in Delaunay
- k-d: O(k n^2). dense Prim

34



Divide and Conquer

35

Divide-and-Conquer

Divide-and-conquer.

- Break up problem into several parts.
- Solve each part recursively.
- Combine solutions to sub-problems into overall solution.

Most common usage.

- Break up problem of size n into two equal parts of size $\frac{1}{2}$n.
- Solve two parts recursively.
- Combine two solutions into overall solution in linear time.

Consequence.

- Brute force: n^2.
- Divide-and-conquer: n log n.

> Divide et impera.
> Veni, vidi, vici.
> - Julius Caesar

36

5.1 Mergesort

Sorting

Sorting. Given n elements, rearrange in ascending order.

Applications.
- Sort a list of names.
- Organize an MP3 library.
- Display Google PageRank results.
- List RSS news items in reverse chronological order.

obvious applications

- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

problems become easy once items are in sorted order

- Data compression.
- Computer graphics.
- Computational biology.
- Supply chain management.
- Book recommendations on Amazon.
- Load balancing on a parallel computer.

. . .

non-obvious applications

38

Mergesort

Mergesort.
- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.

Jon von Neumann (1945)

A L G O R I T H M S

A L G O R   I T H M S   divide O(1)

A G L O R   H I M S T   sort 2T(n/2)

A G H I L M O R S T   merge O(n)

$$T(n) \le 2\,T\left(\frac{n}{2}\right) + O(n)$$

39

Merging

Merging. Combine two pre-sorted lists into a sorted whole.

How to merge efficiently?
- Linear number of comparisons.
- Use temporary array.

05demo-merge.ppt

~~A~~ ~~G~~ ~~L~~ O R   ~~H~~ ~~I~~ M S T

A G H I

Challenge for the bored. In-place merge. [Kronrud, 1969]

using only a constant amount of extra storage

40

A Useful Recurrence Relation

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

Mergesort recurrence.

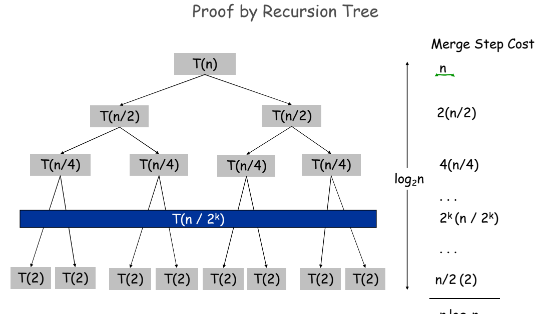
$$T(n) \le \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

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 ≤ with =.

41

Proof by Recursion Tree



$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

42

Proof by Telescoping

Claim. If T(n) satisfies this recurrence, then T(n) = n $\log_2$ n.

↑ assumes n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. For n > 1:

$$\begin{aligned} \frac{T(n)}{n} &= \frac{2T(n/2)}{n} + 1 \\ &= \frac{T(n/2)}{n/2} + 1 \\ &= \frac{T(n/4)}{n/4} + 1 + 1 \\ &\cdots \\ &= \frac{T(n/n)}{n/n} + \underbrace{1 + \cdots + 1}_{\log_2 n} \\ &= \log_2 n \end{aligned}$$

43

Proof by Induction

Claim. If T(n) satisfies this recurrence, then T(n) = n $\log_2$ n.

↑ assumes n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. (by induction on n)
- Base case: n = 1.
- Inductive hypothesis: T(n) = n $\log_2$ n.
- Goal: show that T(2n) = 2n $\log_2$ (2n).

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2n\log_2 n + 2n \\ &= 2n\big(\log_2(2n) - 1\big) + 2n \\ &= 2n\log_2(2n) \end{aligned}$$

44

Analysis of Mergesort Recurrence

Claim. If T(n) satisfies the following recurrence, then T(n) $\leq$ n$\lceil \lg n \rceil$.

↑ $\log_2 n$

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

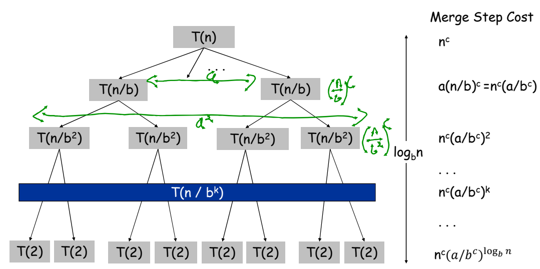
Pf. (by induction on n)
- Base case: n = 1.
- Define $n_1 = \lfloor n / 2 \rfloor$, $n_2 = \lceil n / 2 \rceil$.
- Induction step: assume true for 1, 2, ... , n–1.

$$\begin{aligned} T(n) &\le T(n_1) + T(n_2) + n \\ &\le n_1\lceil \lg n_1 \rceil + n_2\lceil \lg n_2 \rceil + n \\ &\le n_1\lceil \lg n_2 \rceil + n_2\lceil \lg n_2 \rceil + n \\ &= n\lceil \lg n_2 \rceil + n \\ &\le n(\lceil \lg n \rceil - 1) + n \\ &= n\lceil \lg n \rceil \end{aligned}$$

$$\begin{aligned} n_2 &= \lceil n/2 \rceil \\ &\le \left\lceil 2^{\lceil \lg n \rceil} / 2 \right\rceil \\ &= 2^{\lceil \lg n \rceil} / 2 \\ \Rightarrow \lg n_2 &\le \lceil \lg n \rceil - 1 \end{aligned}$$

45

More General Analysis

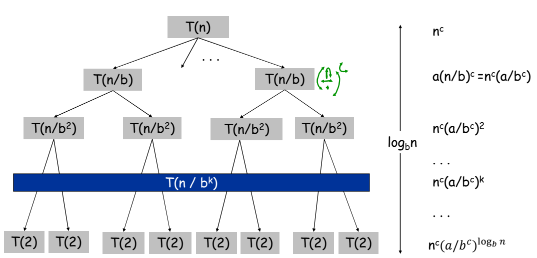


$$T(n) \le \begin{cases} 1 & \text{if } n = 1 \\ a \times T\left(\frac{n}{b}\right) + n^c & \text{otherwise} \end{cases}$$

$$T(n) \le \sum_{i=0}^{\log_b n} n^c \left(\frac{a}{b^c}\right)^i$$

46

More General Analysis



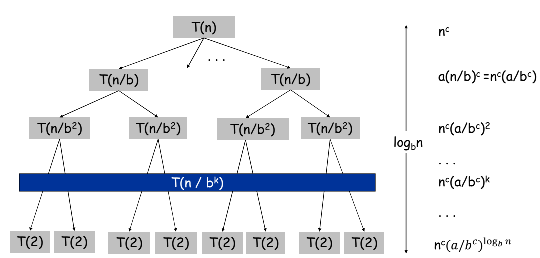
$$T(n) \le \begin{cases} 1 & \text{if } n = 1 \\ a \times T\left(\frac{n}{b}\right) + n^c & \text{otherwise} \end{cases}$$

Case 1: $\gamma = \left(\frac{a}{b^c}\right) = 1$

$$T(n) \le \sum_{i=0}^{\log_b n} n^c \left(\frac{a}{b^c}\right)^i = n^c \log_b n$$

47

More General Analysis



$$T(n) \le \begin{cases} 1 & \text{if } n = 1 \\ a \times T\left(\frac{n}{b}\right) + n^c & \text{otherwise} \end{cases}$$

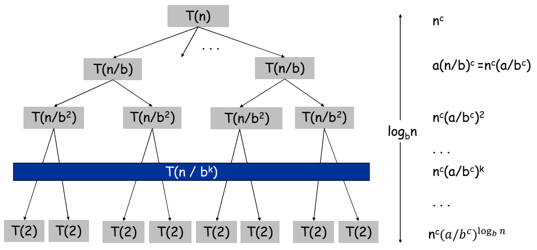
Case 2: $\gamma = \left(\frac{a}{b^c}\right) < 1$

$$T(n) \le \sum_{i=0}^{\log_b n} n^c \left(\frac{a}{b^c}\right)^i = \Theta(n^c)$$

$$n^c \frac{(1-\gamma)}{(1-\gamma)} \sum_{i=0}^{\infty} \gamma^i = \frac{n^c}{(1-\gamma)}$$

48

More General Analysis



$$T(n) \le \begin{cases} 1 & if \\ a \times T\left(\frac{n}{b}\right) + n^c & \end{cases}$$

Case 3: $\left(\frac{a}{b^c}\right) > 1$

$$T(n) \le \sum_{i=0}^{\log_b n} n^c \left(\frac{a}{b^c}\right)^i = \Theta\left(n^{\log_b a}\right)$$

49