1 Dynamic Programming

1.1 Knapsack Problem

Given a set of n items with integer sizes s_i and values v_i, and a knapsack of capacity C, find a set of items whose total size is less than C, but whose value is maximized. Items may be used more than once.

Subproblem: $M(j)$: the maximum value that can be packed into a size j knapsack

Recurrence: $M(j) = \max\{M(j-1), \max_{1\leq i\leq n} M(j-s_i)+v_i\}$

Running time: $O(nC)$ time: C subproblems, each taking $O(n)$ time.

1.2 0/1 Knapsack Problem

The knapsacp problem, but it is forbidden to use more than one of each item.

Subproblem: $M(i,j)$: optimal value for filling *exactly* capacity j knapsack with a subset of items $1,\ldots,i$.

Recurrence: $M(i,j) = \max\{M(i-j,j), M(i-1,j-s_i)+v_i\}$

These represent not using the ith item, and using the ith item, respectively.

Solution: $\max_j\{M(n,j)\}$

Running time: $O(nC)$ time: nC subproblems that take $O(1)$ time

1.3 Longest Path in a DAG

Given a DAG $G=(V,E)$, find the longest $s\to t$ path.

Subproblem: $d[u]$: length of longest $u\to t$ path

Recurrence: $d[u] = \max_{(u,v)\in E}(w(u,v)+d[v])$

Running Time: $\Theta(V+E)$: Solve with a DFS on G.

1.4 Max Value Contiguous Subsequence

Given a sequence of n real numbers $A_1,\ldots,A_n$, determine a contiguous subsequence $A_i,\ldots,A_j$ for which the $\sum_{l=i}^{j} A_l$ is maximized.

Subproblem: $M(j)$: max window ending at j

Recurrence: $M(j) = \max\{M(j-1)+A_j, A_j\}$

Solution: $\max_j M(j)$

Running time: $O(n)$ time: n subproblems each take $O(1)$ time

1.5 Making Change

Given coin denominations of values $1 = v_1 < \ldots < v_n$, and an amount of money C, make C cents in change with as few coins as possible.

Subproblem: $M(j)$: minimum coins needed to make change for j cents

Recurrence: $M(j) = \min_i\{M(j-v_i)\}+1$

Running time: $O(nC)$ time: C subproblems that take $O(n)$ time

1.6 Longest Increasing Subsequence

Given a sequence of n real numbers $A_1,\ldots,A_n$, find the longest subsequence (not necessarily contiguous) where values in the subsequence are strictly increasing.

Subproblem: $L(j)$: longest strictly increasing subsequence ending at position j.

Recurrence: $L(j) = \max_{i<j, A_i<A_j}\{L(i)\}+1$

Solution: $\max_j L(j)$

Running time: $O(n^2)$ time: n subproblems that each take n time

1.7 Edit Distance

Given strings A of length n and B of length m, transform A into B using a series of inserts, deletes, and replacements, of cost C_i, C_d, and C_r, minimizing the cost.

Subproblem: $T(i,j)$: minimum cost to transform $A[:i]$ into $B[:j]$.

Recurrence:

$$T(i,j) = \min\begin{cases} C_d + T(i-1,j) \\ T(i,j-1)+C_i \\ \begin{cases} T(i-1,j-1), A[i]==B[j] \\ T(i-1,j-1), \text{else} \end{cases} \end{cases}$$

Running time $O(nm)$ time: nm subproblems that each take $O(1)$ time

1.8 Counting Boolean Parenthizations

Given a boolean expression consisting of n `True`s and `False`s connected by either `and`, `or`, or `xor`, find the number of ways to parenthesize the expression such that it evaluates to `True`.

Subproblems $T(i,j)$: Number of ways to parenthesize booleans i through j such that the subexpression evaluates to `True`.

$F(i,j)$: similar, but number of ways to make subexpression `False`

$\mathrm{S}(i,j) = T(i,j)+F(i,j)$

Recurrence

$$T(i,j) = \sum_{i\leq k\leq j-1}\begin{cases} T(i,k)T(k+1,j) \\ \text{operator after } k \text{ is } \texttt{and} \\ \mathrm{S}(i,k)\,\mathrm{S}(k+1,j) - F(i,k)F(k+1,j) \\ \text{operator after } k \text{ is } \texttt{or} \\ T(i,k)F(k+1,j)+F(i,k)T(k+1,j) \\ \text{operator after } k \text{ is } \texttt{xor} \end{cases}$$

Running time: $O(n3)$: n^2 subproblems, each taking $O(n)$ time

1.9 Subset Sum

Given a sequence of n numbers $x_1,\ldots,x_n$, select the subset with the maximum total sum, subject to the constraint that you can't select two adjacent elements

Subproblems $S(i)$: maximum sum of the first i numbers

Recurrence $S(i) = \max(S(i-2)+x_i, S(i))$

Running time: $O(n)$ time: n subproblems, each taking $O(1)$ time.

1.10 Reducable Problems

1.10.1 Box Stacking

Given a stack of 3-D boxes with height h_i, width w_i, and depth d_i, create the tallest stack of boxes possible, where box j can only be stacked on box i if $w_i > w_j$ and $d_i > d_j$.

Reduction: Make a box for each possible rotation of the original set of boxes. Assume $w_i \leq d_i$. Sort boxes in order of decreasing base area. Problem is now Longest Increasing Subsequence

1.10.2 Building Bridges

Consider a map with a horizontal river. There are n cities on the southern bank with increasing x-coordinates $A_1,\ldots,A_n$, and correponding cities on the northern bank with x-coordinates (that are not necessarily increasing) $B_1,\ldots,B_n$. Connect as many A_i, B_i pairs as possible without bridges crossing.

Reduction: Define a list with element i as the index of city i on the northern bank. Find the Longest Increasing Subsequence of that list.

1.10.3 Picking Up Pennies

Given a DAG $G=(V,E)$ where some edges have pennies, find a path $s\to t$ with the most pennies.

Reduction: Define $w(u,v)$ = number of pennies on edge. Then Longest Path.

2 Asymptotics

$f = O(g) \iff \lim_{n\to\infty} f(n)/g(n) \neq \infty$
$f = \Omega(g) \iff \lim_{n\to\infty} f(n)/g(n) \neq 0$
$f = \Theta(g) \iff \lim_{n\to\infty} f(n)/g(n) \notin \{0,\infty\}$
$f = \Theta(g) \iff f = O(g)$ and $f = \Omega(g)$
$f = \Theta(g) \iff g = \Theta(f)$
$f = O(g) \iff g = \Theta(f)$

2.1 Common Functions

Ordered from slowest growing to fastest growing.

1	$\log\log n$
$\log n$	$(\log n)^c, c>1$
$n^c, 0<c<1$	n
$n\log n = \log n!$	n^2
n^3	$n^c, c>0$
2^n	3^n
$c^n, c>1$	$n!$

3 Heaps

Invariant (for max-heap): if B is a child of A then $A > B$.

- `heapify` $\Theta(\log n)$

  Assumes that `LEFT(i)` and `RIGHT(i)` are heaps, but that $A[i]$ may violate the heap invariant

```
def heapify(A, i):
    l = left(i)
    r = right(i)
    if l <= len(A) and A[l] > A[i]:
        largest = l
    else:
        largest = i

    if r <= len(A) and A[r] > A[largest}:
        largest = r

    if largest != i:
        A[i], A[largest] = A[largest], A[i]
        heapify(A, largest)
```

- `extract-max` $\Theta(\log n)$

```
def extract_max(A):
    max = A[0]
    A[0] = A.pop(-1)
    heapify(A, 0)
    return max
```

- `increase-key` $\Theta(\log n)$

```
def increase_key(A, i, key):
    A[i] = key
    while i > 0 and A[parent(i)] < A[i]:
        A[i], A[parent(i)] = A[parent(i)], A[i]
        i = parent(i)
```

- `insert` Add $-\infty$ to the end of the heap, then `increase-key` to the actual value

4 Sorting

Algorithm	Best	Worst	Stable	In-place
Insertion Sort	n	n^2	Y	Y
Merge Sort	$n \lg n$	$n \lg n$	Y	N
Heap Sort	n	$n \lg n$	N	Y
Counting Sort	n	n	Y	N
Quicksort	$n \lg n$	n^2	some versions	N

- Insertion Sort

  Very good on sorted sets; online; good for small sets. Atrocious for large sets.

```
INSERTION-SORT(A)
for j <- 2 to length[A]
    do key <- A[j]
        Insert A[j] into the sorted sequence A[1... j - 1].
        i <- j - 1
        while i > 0 and A[i] > key
            do A[i + 1] <- A[i]
               i <- i - 1
        A[i + 1] <- key
```

- Merge Sort

  Wikipedia: "Although heapsort has the same time bounds as merge sort, it requires only $\Theta(1)$ auxiliary space instead of merge sort's $\Theta(n)$, and is often faster in practical implementations. Quicksort, however, is considered by many to be the fastest general-purpose sort algorithm. On the plus side, merge sort is a stable sort, parallelizes better, and is more efficient at handling slow-to-access sequential media. Merge sort is often the best choice for sorting a linked list: in this situation it is relatively easy to implement a merge sort in such a way that it requires only $\Theta(1)$ extra space, and the slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible."

```
MERGE-SORT(A, p, r)
if p < r
  then q <- (floor((p + r)/2)
       MERGE-SORT(A, p, q)
       MERGE-SORT(A, q + 1, r)
       MERGE(A, p, q, r)

function merge(left,right)
    var list result
    while length(left) > 0 and length(right) > 0
        if first(left) <= first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
    end while
    while length(left) > 0
        append left to result
    while length(right) > 0
        append right to result
    return result
```

- Heap Sort

  Algorithm in a nutshell: 1. Build tree. 2. Grab root. 3. Max-heapify. 4. Go to 2. Not stable, but the $n \lg n$ worst case is nice. Usually slower than quicksort.

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i <- length[A] downto 2
   do exchange A[1] <-> A[i]
      heap-size[A] <- heap-size[A] - 1
      MAX-HEAPIFY(A, 1)
```

- Counting Sort

```
COUNTING-SORT(A, B, k)
for i <- 0 to k
   do C[i] <- 0
for j <- 1 to length[A]
   do C[A[j]] <- C[A[j]] + 1
C[i] now contains the number of elements equal \
  to i.
for i <- 1 to k
   do C[i] <- C[i] + C[i - 1]
C[i] now contains the number of elements less \
  than or equal to i.
for j <- length[A] downto 1
   do B[C[A[j]]] <- A[j]
      C[A[j]] <- C[A[j]] - 1
```

- Quicksort

```
QUICKSORT(A, p, r)
if p < r
   then q <- PARTITION(A, p, r)
        QUICKSORT(A, p, q - 1)
        QUICKSORT(A, q + 1, r)

PARTITION(A, p, r)
x <- A[r]
i <- p - 1
for j <- p to r - 1
    do if A[j] <= x
          then i <- i + 1
               exchange A[i] <-> A[j]
exchange A[i + 1] <-> A[r]
return i + 1
```

- Radix sort

```
RADIX-SORT(A, d)
for i <- 1 to d
   do use a stable sort to sort array A on digit i
```

  Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d(n+k))$ time. Stable; not in-place.

- **Bucket Sort**

  Assumes that the input is generated by a random process that distributes elements uniformly over [0,1)

```
BUCKET-SORT(A)
n <- length[A]
for i <- 1 to n
   do insert A[i] into list B[floor(n A[i])]
for i <- 0 to n - 1
   do sort list B[i] with insertion sort
concatenate the lists B[0], B[1],..., B[n - 1] together in order
```

5 Search

- $d[u]$ is time u is discovered (added to queue)
  $f[u]$ is time that u is finished (removed from queue)
- Tree edges are edges in the depth-first forest G_p. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v). Edge (u,v) is a tree or forward edge iff $d[u] < d[v] < f[v] < f[u]$
- Back edges are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Self-loops, which may occur in directed graphs, are considered to be back edges. Edge (u,v) is a back edge if and only if $d[v] < d[u] < f[u] < f[v]$
- Forward edges are those nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
- Cross edges are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees. Edge (u,v) is a cross edge if and only if $d[v] < f[v] < d[u] < f[u]$.
- Diameter is the maximum-weight shortest path.
- A connected graph has a minumum of $V - 1$ edges and a maximum of $V^2/2$ if undirected, twice that if not. A connected graph has $V^2/2$ if undirected, etc. A sparse graph has fewer than $O(V)$ edges.

5.1 Topological Sort

A topological sort of a dag G = (V, E) is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering. (If the graph is not acyclic, then no linear ordering is possible.) Topological orderings are not necessarily unique

```
TOPOLOGICAL-SORT(G)
call DFS(G) to compute finishing times f[v] for each vertex v
as each vertex is finished, insert it onto the front of a linked list
return the linked list of vertices
```

We can perform a topological sort in time $\Theta(V+E)$, since depth-first search takes $\Theta(V+E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.

5.2 Shortest Path

- **Shortest path - general**
  In computer science, a problem is said to have optimal substructure if an optimal solution can be constructed efficiently from optimal solutions to its subproblems. (In this context: shortest paths between two points have other shortest paths in them)
  If there is a negative-weight cycle reachable from s, however, shortest-path weights are not well defined. No path from s to a vertex on the cycle can be a shortest path–a lesser-weight path can always be found that follows the proposed "shortest" path and then traverses the negative-weight cycle. If there is a negative-weight cycle on some path from s to v, we define $\delta(s, v) = -\infty$
  Triangle inequality: Cutting out an intermediate stop never increases the cost. We formalize this notion by saying that the cost function c satisfies the triangle inequality if for all vertices $u, v, w \in V, c(u, w)c(u, v) + c(v, w)$
  Predecessor subgraph: all $\pi[i]$
- **Relaxation**
  The process of relaxing[1] an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating d[v] and [v]. A relaxation step may decrease the value of the shortest-path esti-mate d[v] and update v's predecessor field $\pi[v]$. The following code performs a relaxation step on edge (u, v).

```
RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
   then d[v] <- d[u] + w(u, v)
        pi[v] <- u
```

- **Bellman-Ford**
  The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph G = (V, E) with source s and weight function $w : E \to R$, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights. If the graph does contain a cycle of negative weights, Bellman-Ford can only detect this; Bellman-Ford cannot find the shortest path that does not repeat any vertex in such a graph. This problem is at least as hard as the NP-complete longest path problem.
  The algorithm uses relaxation, progressively decreasing an estimate d[v] 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 $\delta(s, v)$. The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i <- 1 to |V[G]| - 1
3       do for each edge (u, v) in E[G]
4              do RELAX(u, v, w)
5  for each edge (u, v) in E[G]
6       do if d[v] > d[u] + w(u, v)
7             then return FALSE
8  return TRUE
```

  The Bellman-Ford algorithm runs in time $O(VE)$, since the initialization in line 1 takes $\Theta(V)$ time, each of the $|V|$ - 1 passes over the edges in lines 24 takes $\Theta(E)$ time, and the for loop of lines 57 takes $O(E)$ time.
- **Dijkstra**
  Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph G = (V, E) for the case in which all edge weights are nonnegative. In this section, therefore, we assume that w(u, v) ≤ 0 for each edge (u, v) $\in$ E. As we shall see, with a good implementation, the running time of Dijkstra's algorithm is lower than that of the Bellman-Ford algorithm.
  Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex u V - S with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.
  Runtime:
  - If we simply store d[v] in the vth entry of an array. Each INSERT and DECREASE-KEY operation takes O(1) time, and each EXTRACT-MIN operation takes O(V) time (since we have to search through the entire array), for a total time of $O(V^2 + E) = O(V^2)$.
  - If the graph is sufficiently sparsein particular, $E = o(V^2/\lg V)$it is practical to implement the min-priority queue with a binary min-heap. Each EXTRACT-MIN operation then takes time $O(\lg V)$. As before, there are $|V|$ such operations. The time to build the binary min-heap is $O(V)$. Each DECREASE-KEY operation takes time $O(\lg V)$, and there are still at most $|E|$ such operations. The total running time is therefore $O((V + E)lgV)$, which is $O(ElgV)$ if all vertices are reachable from the source. This running time is an improvement over the straightforward $O(V^2)$-time implementation if $E = o(V^2/\lg V)$.
  - We can in fact achieve a running time of $O(V \lg V + E)$ by implementing the min-priority queue with a Fibonacci heap (see Chapter 20). The amortized cost of each of the —V— EXTRACT-MIN operations is $O(\lg V)$, and each DECREASE-KEY call, of which there are at most $|E|$, takes only $O(1)$ amortized time.

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S <- null
Q <- V[G]
while Q != null
    do u <- EXTRACT-MIN(Q)
       S <- S {u}
       for each vertex v in Adj[u]
           do RELAX(u, v, w)
```

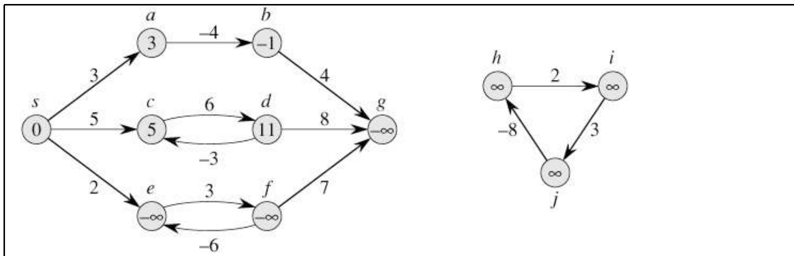


Figure 24.1: Negative edge weights in a directed graph. Shown within each vertex is its shortest-path weight from source s. Because vertices e and f form a negative-weight cycle reachable from s, they have shortest-path weights of $-\infty$. Because vertex g is reachable from a vertex whose shortest-path weight is $-\infty$, it, too, has a shortest-path weight of $-\infty$. Vertices such as h, i, and j are not reachable from s, and so their shortest-path weights are $-\infty$, even though they lie on a negative-weight cycle.

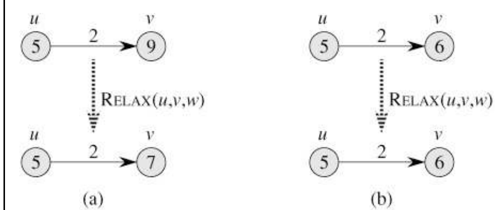


Figure 24.3: Relaxation of an edge (u, v) with weight w(u, v) = 2. The shortest-path estimate of each vertex is shown within the vertex. (a) Because d[v] > d[u] + w(u, v) prior to relaxation, the value of d[v] decreases. (b) Here, d[v] $\leq$ d[u] + w(u, v) before the relaxation step, and so d[v] is unchanged by relaxation.

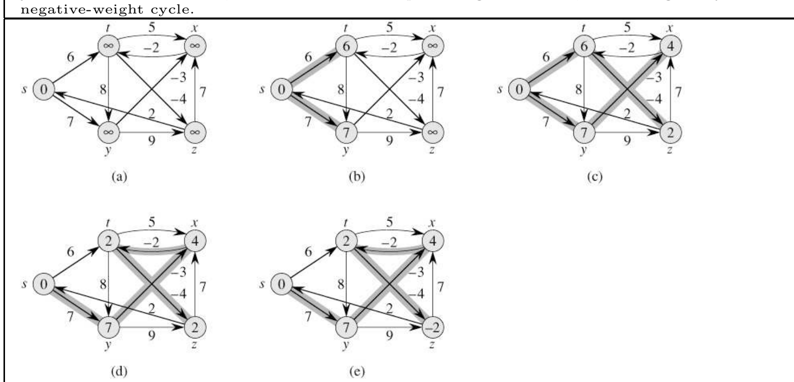


Figure 24.4: The execution of the Bellman-Ford algorithm. The source is vertex s. The d values are shown within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $\pi[v] = u$. In this particular example, each pass relaxes the edges in the order (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y). (a) The situation just before the first pass over the edges. (b)(e) The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

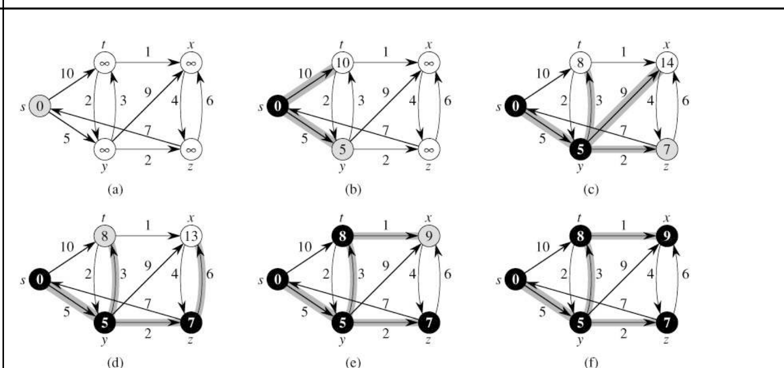


Figure 24.6: The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue Q = V - S. (a) The situation just before the first iteration of the while loop of lines 48. The shaded vertex has the minimum d value and is chosen as vertex u in line 5. (b)-(f) The situation after each successive iteration of the while loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d and values shown in part (f) are the final values.

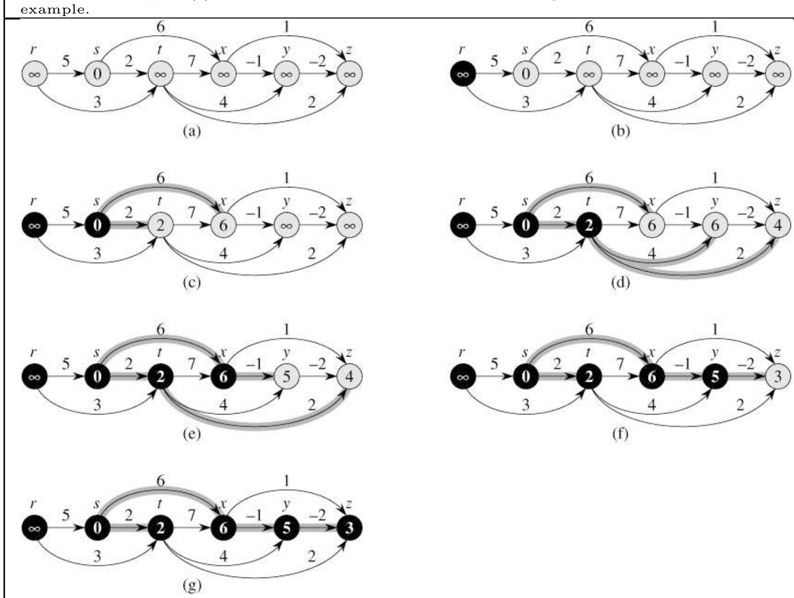


Figure 24.5: The execution of the algorithm for shortest paths in a directed acyclic graph. The vertices are topologically sorted from left to right. The source vertex is s. The d values are shown within the vertices, and shaded edges indicate the values. (a) The situation just before the first iteration of the for loop of lines 35. (b)(g) The situation after each iteration of the for loop of lines 35. The newly blackened vertex in each iteration was used as u in that iteration. The values shown in part (g) are the final values.

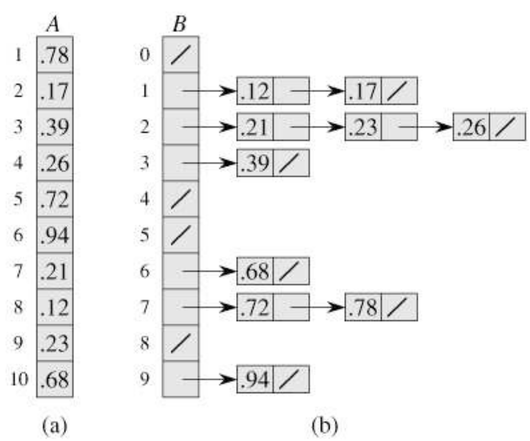
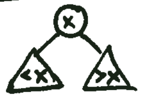
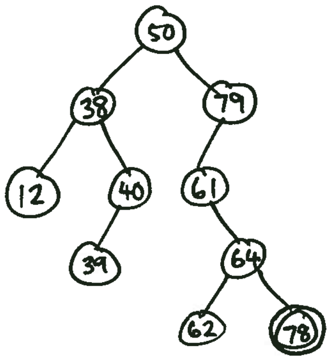


Figure 8.4: The operation of BUCKET-SORT. (a) The input array A[1 10]. (b) The array B[0 9] of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the half-open interval [i/10, (i + 1)/10). The sorted output consists of a concatenation in order of the lists B[0], B[1], . . ., B[9].

TREES/BSTs





A. INSERT(78) $\Theta(h)$

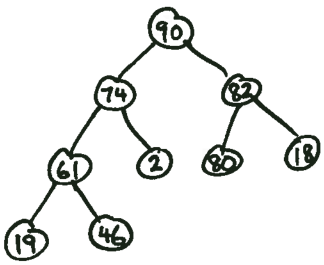
B. DELETE(50) — swap with 61 and delete $\Theta(h)$

C. SUCCESSOR $\Theta(h)$

D. IN-ORDER WALK $\Theta(n)$

– AND: CONSTRUCTING A BST FROM A SORTED LIST... (ALSO $\Theta(n)$)

HEAPS



NEARLY-COMPLETE BINARY TREE

← MAX-HEAP... OF COURSE, MIN-HEAPS EXIST, TOO.

B. EXTRACT-MIN* $\Theta(\lg n)$

*for min-heaps.

A. HEAPIFY $\Theta(\lg n)$

C. BUILD-HEAP $\Theta(n)$

D. PRIORITY QUEUE

– DECREASE-KEY*

ARRAY REPRESENTATION:

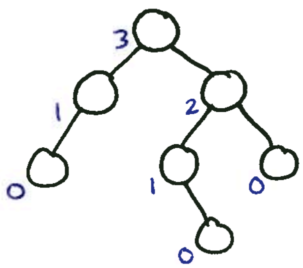
90	74	82	61	2	- ...
0	1	2	3	4	...

PARENT[j] = [redacted] $\left\lfloor \frac{j-1}{2} \right\rfloor$

CHILD[i]:

LEFT[i] = $2i + 1$ RIGHT[i] = $2i+2$

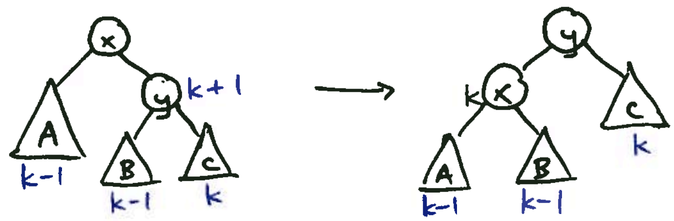
AVL TREES: BALANCED BSTs



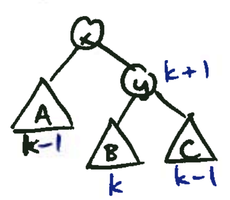
$1 \geq |h(\text{LEFT}) - h(\text{RIGHT})|$

REBALANCING ONLY REQUIRES ROTATIONS

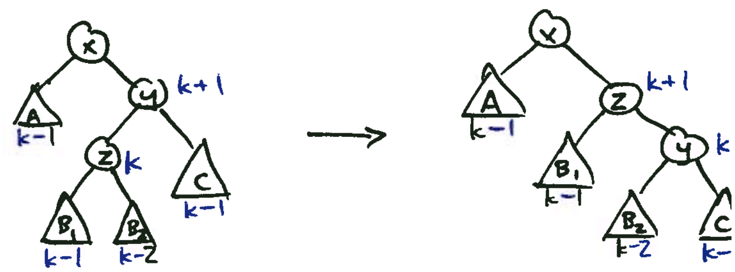
EASY CASE:



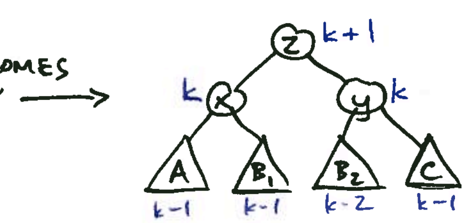
HARD CASE:



THE ABOVE WON'T WORK... WE NEED TWO ROTATIONS

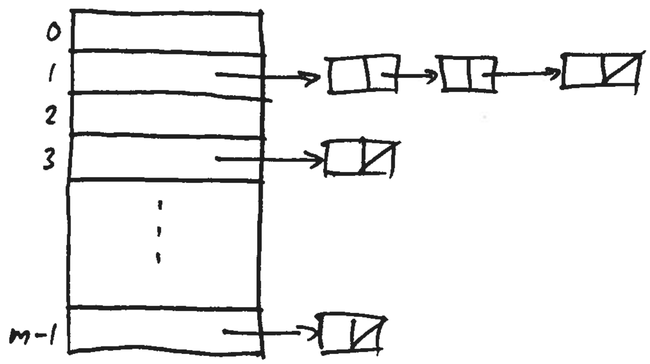


THEN IT BECOMES THE EASY CASE...



006 REVIEW SESSION — DECEMBER 11, 2008

HASHING / HASH TABLES



A. COLLISION RESOLUTION

B. INSERT

C. DELETE

D. LOOKUP

B–D: EXPECTED $\Theta(1)$, WORST CASE $\Theta(n)$

E. ROLLING HASHES

LOAD FACTOR $\alpha = \frac{n}{m}$

(n ELTS. INSERTED)

AMORTIZED ANALYSIS

HASH TABLE (IGNORE DELETIONS) — WANT TO KEEP $\alpha < 4/5$

ON INSERT, IF $\alpha \geq 4/5$, ALLOCATE A NEW TABLE OF $2m$ SIZE AND REHASH EVERYTHING

$\rightarrow O(m+n) = O(n)$ TIME TO RESIZE

$O(1)$ OTHERWISE

IN A SERIES OF n INSERTIONS:

- suppose resizes occur at $k, 2k, 4k, \ldots$
- then cost of insertions is $O(n + (k + 2k + \ldots + n)) = O(n + (\frac{n}{2^i} + \frac{n}{2^{i-1}} + \cdots + n))$

$$= O(n + (2n))$$

$$= O(3n) = O(n)$$

Thus each insert is $\frac{O(n)}{n} = O(1)$ time