Drive from BU CS to Logan Airport

Single-destination shortest paths problem.
Shortest paths in weighted graphs

Input.

• Directed or undirected graph $G(V,E)$ (today: directed)

• (optional) source node $s$, (optional) destination node $t$

• Weight $l(u,v)$ on every edge $(u,v)$
  - $l(u,v)$ can take any value, e.g. positive or negative, integer or real
  - referred to as “length”, “weight”, “cost” depending on the application
  - (we’re sometimes going to be sloppy and use these words interchangable)

• Length, weight or cost of a path $(v_1, v_2, …, v_k)$ is the sum of values $l(v_i,v_{i+1})$ along the edges.

$$\text{length} = \sum_{i=1}^{k-1} l(v_i, v_{i+1})$$

**Length ≠ Number of Edges**

Shortest paths problem: Find a path between two nodes of minimum total weight
Seam Carving example

original - Broadway tower, Cotswolds, England

https://en.wikipedia.org/wiki/Seam_carving
Seam Carving example

original

scaled

https://en.wikipedia.org/wiki/Seam_carving
Seam Carving example

original

scaled

cropped

https://en.wikipedia.org/wiki/Seam_carving
Seam Carving example

original

paths of least significant pixels

[Image: https://en.wikipedia.org/wiki/Seam_carving]
Seam Carving example

original

paths of least significant pixels

final

https://en.wikipedia.org/wiki/Seam_carving
Single source shortest paths — Dijkstra’s algorithm

Input.
• Directed graph $G(V,E)$
• Edge lengths $l(u,v) \geq 0$
• source $s$

Return.
• Distance from $s$ to every node
• Parent table - shortest paths tree from $s$ to each node

Dijkstra’s only works with non-negative edge weights.
Subpaths of a shortest paths

**proposition.** Suppose that there is a shortest paths from $u$ to $v$. Then any subpath between nodes $x$ and $y$ on this path is a shortest path from $x$ to $y$.

**proof:**

Suppose there is another path from $x$ to $y$. If we replace the section of the $u \to v$ path between $x$ and $y$ with this shorter path, then the overall path would be shorter. This contradicts the fact that we had a shortest path from $u$ to $v$. 
Idea: parts of a shortest paths are also shortest paths

**Observation.** The shortest path from $s$ to $v$ will contain one of the edges $(x,v)$, $(y,v)$ or $(z,v)$

some directed path from $s$ to $x$
(may consist of multiple edges)
**Top Hat Question**

**Question.** Suppose that node \( v \) has 3 incoming edges \((x,v), (y,v)\) and \((z,v)\). Given the distance from \( s \) to \( x, y, z \) and the weights on each edge, what is \( \text{dist}(s,v) \)?

\[
\text{dist}(s,v) = \min \left\{ \text{dist}(s,u) + \lambda(u,v) \right\}_{u \in \{x, y, z\}}
\]

- A. \( \text{dist}(s,v) = 7 \)
- B. \( \text{dist}(s,v) = 8 \)
- C. \( \text{dist}(s,v) = 2 \)
- D. \( \text{dist}(s,v) = 4 \)
Dijkstra’s algorithm — insight

Let $\text{dist}(s,u)$ denote the shortest path length from $s$ to $u$.

**Claim.** Suppose we know $\text{dist}(s,u)$. Further, there is an edge $(u,v)$ with length $l(u,v)$. Then we know that

$$\text{dist}(s,v) \leq \text{dist}(s,u) + l(u,v)$$

upper bound of the shortest path
Top Hat Question

Question. Suppose that node $v$ has 3 incoming edges $(x,v)$, $(y,v)$ and $(z,v)$. Given the distance from $s$ to $x$, $y$, $z$ and the weights on each edge, which one is the correct formula to compute $\text{dist}(s,v)$?

\[ \text{dist}(s,x) = 8 \]
\[ \text{dist}(s,y) = 4 \]
\[ \text{dist}(s,z) = 4 \]

A. $\text{dist}(s,v) = \min_{u: \text{edge}(u,v)} \text{dist}(s,u) + \ell(u,v)$

B. $\text{dist}(s,v) = \min_{u: \text{edge } (u,v)} \ell(u,v)$

C. $\text{dist}(s,v) = \sum_{u: \text{edge } (u,v)} \ell(u,v)$

D. $\text{dist}(s,v) = \min_{u: \text{edge } (u,v)} \text{dist}(s,u)$
Dijkstra’s algorithm overview

For each node \( v \) we maintain the min length of path we know so far from \( s \) to \( v \).

- this is the *best known* upper bound on dist\((s,v)\) so far
- denoted by \( \pi(v) \) — *this is a number, not a path*

Initialize: for each \( v \) \( \pi(v) = \infty \)

In each iteration:
- find \( u \) with the lowest \( \pi(u) \)
- fix the distance \( dist(s,u) \) to be \( dist(s,u) = \pi(u) \)
- for each neighbor \( v \) of \( u \), update their best known path

\[
\pi(v) = \min\{\pi(v), dist(s,u) + l(u,v)\}
\]

Compare the best upper bound so far against the path that contains the edge \((u,v)\)
Dijkstra’s algorithm example

\[ d(u) = \text{distance from } s \text{ to } u \]
\[ \pi(v) = \text{ currently known shortest distance to } v \]

<table>
<thead>
<tr>
<th>node v</th>
<th>( \pi(v) )</th>
<th>parent</th>
<th>( \text{dist}(s,v) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>None</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
<td>C</td>
<td>7</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>9</td>
<td>B</td>
<td>9</td>
</tr>
<tr>
<td>E</td>
<td>5</td>
<td>C</td>
<td>5</td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm example

\[ d(u) = \text{distance from s to } u \]
\[ \pi(v) = \text{currently known shortest distance to } v \]

Initialize:

- \( s = A \)
- Maintain data structure \( Q \)
- Initially for every \( v \) set \( \pi(v) = \infty \)

\[
\begin{array}{cccccc}
Q & A & B & C & D & E \\
\pi(v) & 0 & \infty & \infty & \infty & \infty & \infty \\
\end{array}
\]

\[ D = \{ d(A) = 0 \} \]
Dijkstra’s algorithm example

\[ d(u) = \text{distance from } s \text{ to } u \]

\[ \pi(v) = \text{currently known shortest distance to } v \]

Initialize:

- \( s = A \)
- Maintain data structure \( Q \)
- Initially for every \( v \) set \( \pi(v) = \infty \)

\[
\begin{array}{cccccc}
Q & A & B & C & D & E \\
\pi(v) & 0 & \infty & \infty & \infty & \infty \\
\end{array}
\]

\[ D = \{d(A) = 0\} \]
Dijkstra’s algorithm example

\[ d(u) = \text{distance from s to u} \]
\[ \pi(v) = \text{currently known shortest distance to v} \]

For which nodes can we update their tentative distance \( \pi(v) \)?

To update compute

\[ \pi(w) = \min(\pi(w), \pi(v) + l(v, w)) \]

\[
\begin{array}{cccccc}
  & Q & A & B & C & D & E \\
\pi(v) & 0 & \infty & \infty & \infty & \infty & \infty \\
\end{array}
\]

\[ D = \{ d(A) = 0 \} \]
Dijkstra’s algorithm example

\[ d(u) = \text{distance from s to } u \]
\[ \pi(v) = \text{currently known shortest distance to } v \]

For which nodes can we update their tentative distance \( \pi(v) \) ?

To update compute

\[ \pi(w) = \min(\pi(w), \pi(v) + l(v, w)) \]

<table>
<thead>
<tr>
<th>Q</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \pi(v) )</td>
<td>0</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>3</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
</tr>
</tbody>
</table>

\[ D = \{d(A) = 0\} \]
Dijkstra’s algorithm example

\[ d(u) = \text{distance from s to u} \]

\[ \pi(v) = \text{currently known shortest distance to v} \]

For which nodes can we update their tentative distance \( \pi(v) \)?

To update compute

\[ \pi(w) = \min(\pi(w), \pi(v) + l(v, w)) \]

<table>
<thead>
<tr>
<th>Q</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \pi(v) )</td>
<td>0</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ D = \{d(A) = 0, d(C) = 3\} \]
Dijkstra’s algorithm example

\[ d(u) = \text{distance from } s \text{ to } u \]
\[ \pi(v) = \text{currently known shortest distance to } v \]

For which nodes can we update their tentative distance \( \pi(v) \)?

To update compute

\[ \pi(w) = \min(\pi(w), \pi(v) + l(v, w)) \]

\[
\begin{array}{cccccc}
Q & A & B & C & D & E \\
\pi(v) & 0 & \infty & \infty & \infty & \infty \\
& 10 & 3 & \infty & \infty \\
& 7 & 11 & 5 & \\
\end{array}
\]

\[ D = \{d(A) = 0, d(C) = 3\} \]
Dijkstra’s algorithm example

\[ d(u) = \text{distance from } s \text{ to } u \]
\[ \pi(v) = \text{currently known shortest distance to } v \]

For which nodes can we update their tentative distance \( \pi(v) \)?

To update compute

\[ \pi(w) = \min(\pi(w), \pi(v) + l(v, w)) \]

Notice, that the only values that (possible) change are for nodes adjacent to \( C \)

\[
\begin{array}{ccccccc}
Q & A & B & C & D & E \\
\pi(v) & 0 & \infty & \infty & \infty & \infty & \infty \\
& 10 & 3 & \infty & \infty & \\
& 7 & 11 & 5 & \\
\end{array}
\]

\[ D = \{ d(A) = 0, d(C) = 3 \} \]
Dijkstra’s algorithm example

\[ d(u) = \text{distance from } s \text{ to } u \]
\[ \pi(v) = \text{currently known shortest distance to } v \]

For which nodes can we update their tentative distance \( \pi(v) \)?

To update compute

\[ \pi(w) = \min(\pi(w), \pi(v) + l(v, w)) \]

<table>
<thead>
<tr>
<th>Q</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \pi(v) )</td>
<td>0</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ D = \{d(A) = 0, d(C) = 3, d(E) = 5\} \]
Dijkstra’s algorithm example

\( d(u) = \text{distance from } s \text{ to } u \)

\( \pi(v) = \text{currently known shortest distance to } v \)

For which nodes can we update their tentative distance \( \pi(v) \) ?

To update compute

\[ \pi(w) = \min(\pi(w), \pi(v) + l(v, w)) \]

<table>
<thead>
<tr>
<th>Q</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \pi(v) )</td>
<td>0</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>3</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>11</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( D = \{ \text{d(A) = 0, d(C) = 3, d(E) = 5} \} \)
Dijkstra’s algorithm example

\( d(u) \) = distance from s to u

\( \pi(v) = \text{currently known shortest distance to } v \)

For which nodes can we update their tentative distance \( \pi(v) \) ?

To update compute

\[
\pi(w) = \min(\pi(w), \pi(v) + l(v, w))
\]

\[\begin{array}{cccccc}
  \pi(v) & 0 & \infty & \infty & \infty & \infty \\
  10 & 3 & \infty & \infty & & \\
  7 & 11 & 5 & & & \\
  7 & & 11 & & & \\
\end{array}\]

\( D = \{d(A) = 0, d(C) = 3, d(E) = 5, d(B) = 7\} \)
Dijkstra’s algorithm example

\[ d(u) = \text{distance from s to u} \]
\[ \pi(v) = \text{currently known shortest distance to v} \]

For which nodes can we update their tentative distance \( \pi(v) \)?

To update compute

\[ \pi(w) = \min(\pi(w), \pi(v) + l(v, w)) \]

\[ Q \]

\begin{array}{cccccc}
\pi(v) & A & B & C & D & E \\
0 & \infty & \infty & \infty & \infty & \infty \\
10 & 3 & \infty & \infty & \infty & \infty \\
7 & 11 & 5 & 11 & 9 & \\
7 & & & & & \\
\end{array}

\[ D = \{d(A) = 0, \ d(C) = 3, \ d(E) = 5, \ d(B) = 7\} \]
Dijkstra’s algorithm example

\[ d(u) = \text{distance from } s \text{ to } u \]

\[ \pi(v) = \text{currently known shortest distance to } v \]

For which nodes can we update their tentative distance \( \pi(v) \) ?

To update compute

\[ \pi(w) = \min(\pi(w), \pi(v) + l(v, w)) \]

\[ Q \quad A \quad B \quad C \quad D \quad E \]

<table>
<thead>
<tr>
<th>\pi(v)</th>
<th>0</th>
<th>\infty</th>
<th>\infty</th>
<th>\infty</th>
<th>\infty</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>3</td>
<td>\infty</td>
<td>\infty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>11</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>11</td>
<td></td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

\[ D = \{d(A) = 0, \ d(C) = 3, \ d(E) = 5, \ d(B) = 7, \ D(d) = 9\} \]
Dijkstra’s algorithm example

\[ d(u) = \text{distance from } s \text{ to } u \]
\[ \pi(v) = \text{currently known shortest distance to } v \]

For which nodes can we update their tentative distance \( \pi(v) \)?

To update compute

\[ \pi(w) = \min(\pi(w), \pi(v) + l(v, w)) \]

Values highlighted in blue are the true \( d(v) \) distances.

\[
\begin{array}{cccccc}
Q & A & B & C & D & E \\
\hline
\pi(v) & 0 & \infty & \infty & \infty & \infty \\
10 & 3 & \infty & \infty & 5 \\
7 & 11 & 11 & 9 \\
\end{array}
\]

\[ D = \{d(A) = 0, d(C) = 3, d(E) = 5, d(B) = 7, D(d) = 9\} \]
Dijkstra’s algorithm

Algorithm 1: Dijkstra(G, s)

1 /* G directed, weighted adjacency list, source s */
2 \( \pi \leftarrow \{ \} /* hash table, current best dist for v */
3 d \leftarrow \{ \} /* hash table, distance of v */
4 parents \leftarrow \{ \} /* hash table, parents in shortest paths tree */
5 for \( v \) in \( G \) do /* We haven’t found a path from \( s \) to \( v \) yet */
6 \( \pi[v] \leftarrow \infty; \)
7 \( \pi[s] \leftarrow 0, parents[s] \leftarrow \text{None}; \)
8 for \( i = 1 \) to \( n \) do
9 \( u \leftarrow \text{unfinished node with min} \ \pi[u]; \)
10 \( d[u] \leftarrow \pi[u] /* fix distance of u */
11 for \( v \) in \( G[u] \) do /* update the distance of neighbors of \( u */
12 if \( \pi[v] > d[u] + G[u][v] \) then
13 \( \pi[v] \leftarrow d[u] + G[u][v]; \)
14 \( parents[v] = u; \)
15 return \( d, parents \)

Questions about the implementation:

• Can we be more efficient about updating the distances? (lines 7-13)
• How do we find the minimum \( u \)? (line 8)
Priority queue — asymptotic running time of operations

In this course, when we refer to a Priority Queue, we always refer to a binary heap implementation. Given that the PQ holds $n$ items the operations take:

- $O(1)$: peaking at the root.
- $O(\log n)$: INSERT, DECREASE-KEY, EXTRACT-MIN (a.k.a. DELETE-MIN)

What this means for the running time of Dijkstra’s:
- the PQ holds entries corresponding to each node of a graph
- during the algorithm we check and update once for every edge
- total time complexity of these operations is $O(m \log n)$.
Dijkstra’s algorithm example — priority queue operations

Initialize Q:

\[ Q.\text{INSERT} \langle 0, A \rangle \]
\[ Q.\text{INSERT} \langle \infty, B \rangle \]
\[ Q.\text{INSERT} \langle \infty, C \rangle \]
\[ Q.\text{INSERT} \langle \infty, D \rangle \]
\[ Q.\text{INSERT} \langle \infty, E \rangle \]

\[ \langle 0, A \rangle = Q.\text{EXTRACT\_MIN()} \]

\[
\begin{array}{cccccc}
\text{Q} & \text{A} & \text{B} & \text{C} & \text{D} & \text{E} \\
\hline
\pi(v) & 0 & \infty & \infty & \infty & \infty
\end{array}
\]
Dijkstra’s algorithm example – priority queue operations

Update distances in Q:

\[ Q.DECREASE\_KEY (<\infty, B>, <10, B>) \]
\[ Q.DECREASE\_KEY (<\infty, C>, <3, C>) \]

Current content of Q:
\[ Q = (<10, B>, <3, C>, <\infty, D>, <\infty, E>) \]
Dijkstra’s algorithm example — priority queue operations

Current content of Q:

\[ Q = (\langle 10, B \rangle, \langle 3, C \rangle, \langle \infty, D \rangle, \langle \infty, E \rangle) \]

\[ \langle 3, C \rangle = Q.\textsc{extract}_\text{MIN}() \]

<table>
<thead>
<tr>
<th>Q</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \pi(v) )</td>
<td>0</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm example — priority queue operations

Update distances in $Q$:

$Q$.DECREASE_KEY ($< 10, B >$, $< 7, B >$)

same for D and E

<table>
<thead>
<tr>
<th>Q</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi(v)$</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>3</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>11</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm

**Algorithm 1: Dijkstra**($G, s$)

```
/* G directed, weighted adjacency list, source s */
1 \( \pi \leftarrow \{ \} */ hash table, current best dist for \( v \) */
2 \( d \leftarrow \{ \} */ hash table, distance of \( v \) */
3 \( \text{parents} \leftarrow \{ \} */ hash table, parents in shortest paths tree */
4 \( Q \leftarrow \text{priority queue} */ keys are current best distances \( \pi[v] \) */
5 \textbf{for} \( v \neq s \) \textbf{in} \( G \) \textbf{do}
6     \( \pi[v] \leftarrow \infty; \)
7     \( Q.\text{INSERT}(< \pi[v], v >); \)
8 \( \pi[s] \leftarrow 0, \text{parents}[s] \leftarrow \text{None}, Q.\text{INSERT}(< \pi[s], s >); \)
9 \textbf{while} \( Q \) \textbf{is not empty} \textbf{do}
10    \( < \pi[u], u > \leftarrow \text{EXTRACT-MIN}(Q); \)
11    \( d[u] \leftarrow \pi[u] /* fix distance of \( u \) */ \)
12 \textbf{for} \( v \) \textbf{in} \( G[u] \) \textbf{do}
13    /* update the distance of neighbors of \( u \)
14    if \( \pi[v] > d[u] + G[u][v] \) then
15        \( \pi[v] \leftarrow d[u] + G[u][v]; \)
16        \( \text{parents}[v] = u; \)
17        \( \text{DECREASE-KEY}(< \pi[v], v >, < d[u] + G[u][v], v >); \)
18 \textbf{return} \( d, \text{parents} \)
```

Overall running time \( O(m \log(n)) \).

You may read online about a running time of \( O(m + n \log(n)) \). This corresponds to an implementation using Fibonacci heaps (we will use \( O(m \log(n)) \)).
Dijkstra’s algorithm: proof of correctness

Invariant. For each node \( u \in S \), \( d(u) \) is the length of a shortest \( s \rightarrow u \) path.
(Note: this implies that once \( u \) is added to \( S \), \( d(u) \) is never changed)

Base case: \(|S| = 1\). Since \( S = \{s\} \), \( d(s) = 0 \).

Inductive hypothesis: Assume that it is true for \(|S| = K > 1\).

Let \( u \) be the next node added to \( S \).
Assume that we have an edge \((x,y)\) and \( P \) is a path between \( y \) and \( u \).

\( P' \) is already too long when it reaches \( y \), because
\( \Pi(u) \leq \Pi(y) \).
Dijkstra’s algorithm: proof of correctness

**Invariant.** For each node \( u \in S \), \( d(u) \) is the length of a shortest \( s \sim u \) path.

**Pf.** [ by induction on \( |S| \) ]

**Base case:** \( |S| = 1 \) is easy since \( S = \{ s \} \) and \( d(s) = 0 \).

**Inductive hypothesis:** Assume true for \( |S| = k \geq 1 \).

- Let \( v \) be next node added to \( S \), and let \( (u, v) \) be the final edge.
- A shortest \( s \sim u \) path plus \( (u, v) \) is an \( s \sim v \) path of length \( \pi(v) \).
- Consider any \( s \sim v \) path \( P \). We show that it is no shorter than \( \pi(v) \).
- Let \( (x, y) \) be the first edge in \( P \) that leaves \( S \), and let \( P' \) be the subpath to \( x \).
- \( P \) is already too long as soon as it reaches \( y \).

\[
\ell(P) \geq \ell(P') + \ell(x, y) \geq d(x) + \ell(x, y) \geq \pi(y) \geq \pi(v)
\]
Dijkstra example
Dijkstra example - negative edge

Dijkstra's fails with negative edge weights. What does that mean?

This is less than the fixed value (look at the previous slide).

Dijkstra’s fails with negative edge weights. What does that mean?
Run Dijkstra’s algorithm from node s. What distance value will the algorithm return for node a and what is the correct length of the shortest path from s to a?

A. $d[a] = 15, \ell(a) = 7$
B. $d[a] = 7, \ell(a) = 7$
C. $d[a] = 2, \ell(a) = 1$
D. $d[a] = 7, \ell(a) = 6$
Review

Question. Suppose that node \( v \) has 3 incoming edges \((x,v), (y,v)\) and \((z,v)\). Given the distance from \( s \) to \( x, y, z \) and the weights on each edge, what is \( \text{dist}(s,v) \)?

![Diagram of a graph with nodes and edges labeled with distances]

\[
\begin{align*}
\text{dist}(s,x) &= 8 \\
\text{dist}(s,y) &= 4 \\
\text{dist}(s,z) &= 4
\end{align*}
\]

Conclusion: the shortest path length can be computed as the minimum over the in-neighbors of \( v \):

\[
\text{dis}(s, v) = \min_{u: \text{edge} \ (u,v)} \{ \text{dist}(s, u) + \ell(u, v) \}
\]

“Greedy” approach: in order for the above to work we need to know for sure that \( \text{dist}(s,u) \) is correct at the time that we use it for \( \text{dist} \) of node \( v \).
Dijkstra’s algorithm overview

For each node \( v \) we maintain the min length of path we know so far from \( s \) to \( v \).

- this is the best known upper bound on \( \text{dist}(s,v) \) so far
- denoted by \( \pi(v) \)

Initialize: for each \( v \) \( \pi(v) = \infty \)

In each iteration:

- find \( u \) with the lowest \( \pi(u) \)
- fix the distance \( \text{dist}(s,u) \) to be \( \text{dist}(s,u) = \pi(u) \)
- for each neighbor \( v \) of \( u \), update their best known path

\[
\pi(v) = \min\{\pi(v), \text{dist}(s,u) + l(u,v)\}
\]

Implementation:

- to compute \( \pi(v) \) instead of taking the minimum over in-neighbors of \( v \)
- we check whether we can update \( \pi(v) \) of the out-neighbors of \( u \) when the distance of a node \( u \) gets fixed.
Dijkstra’s fails with negative edge weights. What does that mean?
Select the true statements for a directed weighted graph G, with no negative edge weight and source s.

A. It’s possible for a node v to have two shortest paths between s to v.

B. If all edge weights are *unique* then the shortest path to each node is *unique*.

C. If there are two different shortest paths from s to v, then Dijkstra’s always finds the one with fewer edges.
History

Philip II of Macedon (BC359)
- divide et impera = divide and rule
- creating or encouraging divisions among the subjects to prevent alliances that could challenge the sovereign

Macchiavelli - The Art of War (1521)
- divide the enemy army into two and then conquer each half one at a time
Divide-and-conquer paradigm

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

Examples:
- mergesort, quicksort, binary search
- geometric problems: convex hull, nearest neighbors
- efficient computations: multiplication of numbers, matrix multiplication
- algorithms for processing on trees
- many data structures: binary search trees, heaps
- parallelizations
Sorting

Input: n numbers
output: n numbers in sorted order

brute-force: all possible orders of n numbers $O(n!) = O(n^n)$

bubble sort: if two neighboring numbers are in opposite order then swap. $O(n^2)$
  • this is already polynomial

all possible permutations!
Divide-and-conquer paradigm - Mergesort

Input: n numbers
output: n numbers in sorted order

one of the earliest sorting algorithms (1945, John von Neumann)

sort by recursive call to algorithm

merge O(n)

divide O(1)
sort 2T(n/2)
Given two sorted lists $A$ and $B$, merge into sorted list $C$.

\[\text{sorted list A}\]
\[
\begin{array}{cccc}
3 & 7 & 10 & 14 & 18 \\
\end{array}
\]

\[\text{size} = \frac{n}{2}\]

\[\text{runtime: } O(n)\]

\[\text{sorted list B}\]
\[
\begin{array}{cccc}
2 & 11 & 16 & 20 & 23 \\
\end{array}
\]

\[\text{size} = \frac{n}{2}\]

\[\text{sorted list C}\]
\[
\begin{array}{cccccccccccccccccccc}
2 & 3 & 7 & 10 & 11 & 14 & 16 & 18 & 20 & 23 \\
\end{array}
\]
# Merge in MergeSort

**Goal.** Combine two *sorted* lists $A$ and $B$ into a sorted whole $C$.

- Scan $A$ and $B$ from left to right.
- Compare $a_i$ and $b_j$.
- If $a_i \leq b_j$, append $a_i$ to $C$ (no larger than any remaining element in $B$).
- If $a_i > b_j$, append $b_j$ to $C$ (smaller than every remaining element in $A$).

**Exercise.** Write the pseudocode for merge

<table>
<thead>
<tr>
<th>sorted list A</th>
<th>sorted list B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 7 10 $a_i$ 18</td>
<td>2 11 $b_j$ 20 23</td>
</tr>
</tbody>
</table>

Merge to form sorted list $C$

| 2 3 7 10 11 |
Analyzing recursive algorithms

- **Correctness** almost always uses strong induction
  - prove correctness of base case (typically: $n < \text{constant}$)
  - for arbitrary $n$:
    - assume algorithm performs correctly on all input sizes $k < n$
    - prove the algorithm is correct on input size $n$

- **Time/space** complexity often uses recurrence
  - structure of recurrence reflects algorithm
Claim. The array returned by MergeSort is sorted

proof. Induction on the length $n$ of the input array

Base case:

- $n=1$ sorted. (or $n=2$, MergeSort correctly puts the smaller of two numbers first)

Inductive hypothesis:

- assume that MergeSort correctly sorts any array of length $k < n$

Prove for $n$:

- MergeSort breaks the problem into two arrays $A$ and $B$ of length $n/2$ each
- By the inductive assumption MergeSort correctly sorts $A$ and $B$ in the recursive calls
- We need to show that the merge step maintains the sorted order
  - $a$ in $A$ and $b$ in $B$ are the current lowest values in their lists
  - Merge selects $a$ if $a \leq b$.
  - $a$ is less than all numbers in $A$, as $A$ is sorted
  - $a$ is less than all in $B$, as $b$ is less than all other elements in $B$
Algorithm 1: MergeSort( \( A, p, r \) )

/* Sorts the subarray \( A[p:r] \) in place */
1. if \( p == r \) then
2.     return \( A \)
3.  \( q \leftarrow \lceil \frac{p+r}{2} \rceil \);
4.  \( A[p:q] \leftarrow \text{MergeSort}(A, p, q) \);
5.  \( A[q+1:r] \leftarrow \text{MergeSort}(A, q+1, r) \);
6.  \( A[p:r] \leftarrow \text{Merge}(A, p, q, r) \);
7.  return \( A \)

Note that Merge is called on two \textit{sorted} lists

- due to recursive call on MergeSort

Recursive, top-down approach

- initial call on array of length \( n \)
- recursive calls deal with the subarray
- each recursive call is on the left and right half of the input
MergeSort — running time

Algorithm 1: MergeSort(A, p, r)

/* Sorts the subarray A[p:r] in place */
1 if p == r then
2     return A
3 q ← \left\lfloor \frac{p+r}{2} \right\rfloor;
4 A[p:q] ← MergeSort(A, p, q);
5 A[q + 1 : r] ← MergeSort(A, q + 1, r);
6 A[p, r] ← Merge(A, p, q, r);
7 return A

O(r-p)

Recursive calls — how many?

Recurrence:
\[ T(n) \leq 2 T(\frac{n}{2}) + \Theta(n) \]
Recurrence

**Def.** \( T(n) = \) worst case running time on an input of size \( n \)

**Recurrence:** \( T(n) \) expressed using a *recursive* function

Mergesort:
1. divide array into two halves
2. recursive calls to mergesort on both halves
3. merge

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

(Should be \( T([n/2]) + T([n/2]) \) but it doesn’t matter asymptotically \( \rightarrow \) we often assume that \( n \) is a power of 2)

(often we omit the base case as for const \( n \) the running time is const)
MergeSort — recurrence

Algorithm 1: MergeSort( A, p, r )
/* Sorts the subarray A[p : r] in place */
1 if p == r then
2 | return A
3 q ← ⌊p+r/2⌋;
4 A[p : q] ← MergeSort(A, p, q);
5 A[q + 1 : r] ← MergeSort(A, q + 1, r);
6 A[p, r] ← Merge(A, p, q, r);
7 return A

Initial call on p = 0 and r = n-1

length of subarray is n/2
O(n) for p=0, r=n-1

running time on input array of length n
running time on input array of length n/2, called on both halves
running time of non-recursive part

\[
T(n) = 2 \cdot T(n/2) + \Theta(n)
\]
Write the recurrences

(You may assume that \( n \) is always a power of 2 or 3 or whatever is needed)

Some algorithm takes as input an array of \( n \) elements. It divides the array into 3 equal parts and calls itself recursively on all 3 parts. Then it performs \( O(n) \) additional computational steps.

\[
T(n) = 3 \cdot T\left(\frac{n}{3}\right) + O(n)
\]

Some other algorithm takes as input an array of \( n \) elements. It divides itself into 2 equal parts and calls itself recursively on one part. It then performs constant many additional operations.

\[
T(n) = T\left(\frac{n}{2}\right) + \Theta(1)
\]

binary search
TopHat — write recurrence

**Question.** Here is a hypothetical algorithm. What is the corresponding recurrence?

An algorithm takes as input \( n \) items. After performing \( n/2 \) comparison, it divides the data into three equal parts. Calls itself *recursively on two of the parts*.

\[
\begin{align*}
A. \quad T(n) &= 2 \cdot T(n/2) + n/2 \\
B. \quad T(n) &= 3 \cdot T(n/2) + \Theta(1) \\
C. \quad T(n) &= 2 \cdot T(n/3) + \Theta(n) \\
D. \quad T(n) &= 3 \cdot T(n/3) + \Theta(1)
\end{align*}
\]

\[
\frac{n}{2} = \Theta(n)
\]

\[
c_1 \cdot \frac{n}{2} \leq \frac{n}{2} \leq c_2 \cdot n
\]

This is true for \( c_1 = c_2 = \frac{1}{2} \).
Question. Here is a hypothetical algorithm. What is the corresponding recurrence?

An algorithm takes as input \( n \) items. It divides the data into *four parts*, *two* of size \( n/3 \) the other *two* of size \( n/6 \). It makes recursive calls on all parts and finally performs one more operation.

A. \[ T(n) = 4T\left(\frac{n}{3}\right) + \Theta(1) \]

B. \[ T(n) = 4T\left(\frac{n}{6}\right) + \Theta(1) \]

C. \[ T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{n}{6}\right) + \Theta(1) \]

D. \[ T(n) = 2T\left(\frac{n}{3}\right) + 2T\left(\frac{n}{6}\right) + \Theta(1) \]
writing recurrences

Write the recurrences (no need to solve) for the following problems:

1. An algorithm takes as input n items. After performing 1 comparison, it divides the data into two equal parts. Calls itself recursively on only one of the parts.
   \[ T(n) = T\left(\frac{n}{2}\right) + O(n) \]

2. An algorithm takes as input n items. It divides the input into 3 equal parts, makes a recursive call on each part. Then combines it in \( O(n) \) time.
   \[ T(n) = 3 \cdot T\left(\frac{n}{3}\right) + O(n) \]

3. An algorithm takes as input n items. It divides the data into parts of size \( \frac{n}{3} \), \( \frac{n}{6} \), \( \frac{3n}{6} \) and makes recursive calls to it. It combines the results in \( O(\log n) \) time.
   \[ T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{3}\right) + T\left(\frac{n}{6}\right) + O(\log n) \]
writing recurrences

Write the recurrences (no need to solve) for the following problems:

1. An algorithm takes as input n items. After performing 1 comparison, it divides the data into two equal parts. Calls itself recursively on only one of the parts.
   \[ T(n) = T(n/2) + O(1) \]

2. An algorithm takes as input n items. It divides the input into 3 equal parts, makes a recursive call on each part. Then combines it in \( O(n) \) time.
   \[ T(n) = 3T(n/3) + O(n) \]

3. An algorithm takes as input n items. It divides the data into parts of size \( n/3, n/6, 3n/6 \) and makes recursive calls to it. It combines the results in \( O(\log n) \) time.
   \[ T(n) = T(n/3) + T(n/6) + T(n/2) + O(\log n) \]
**MergeSort — recurrence**

**Def.** $T(n) = \text{worst case running time on an input of size } n$

**Mergesort recurrence.**

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

**Closed form formula.**

- we don’t know how to interpret the running time from the recursive formula
- we need to find a formula that is a mathematical function of $n$ with no recursive function calls
- solution for mergesort: $T(n) \in \Theta(n \log n)$

**Solving a recurrence.**

- find the formula for $T(n)$
  - multiple methods
- prove by induction that it works
Recursion tree method

- write out tree of recursive calls
- each node gets assigned the work done during that call to the procedure (dividing and combining)
- total work is the sum of work done at all nodes