**Median Selection**

Let’s precisely define the median.

Given a set $S = \{a_1, a_2, \ldots, a_n\}$, the median is the $k$th smallest element, where $k = \lceil n/2 \rceil$:
- $k = n/2$, if $n$ even
- $k = (n+1)/2$, if $n$ odd

More generally: find $k$th smallest number. Complexity?

- $k = 1$: min $O(n)$
- $k = n$: max $O(n)$
- $k = n/2$: ???

$O(n \log n)$ by sorting

Can do deterministic $O(n)$ (Blum, Floyd, Pratt, Rivest, Tarjan ’72)

Today: randomized (similar idea). Assume no duplicates.

**Divide and Conquer: Choose a Splitter**

- Choose a splitter (pivot) $a_i \in S$
- Find elements smaller and larger than splitter:
  - $S^- = \{a_j : a_j < a_i\}$, $S^+ = \{a_j : a_j > a_i\}$

Cases:
- $|S^-| = k − 1$: $a_i$ is the sought element, done
- $|S^-| \geq k$: recurse on $(S^-, k)$
- $|S^+| < k − 1$: recurse on $(S^+, k − (|S| + 1))$

**How to choose splitter?**

Would it help to land “close” to the sought element?

Not if we land inside the larger of the two sets.

Work done (splitting) is proportional to size of whole set

⇒ Want recursive calls to work on (much) smaller sets

- Best splitter is the median!
  
  $T(n) \leq T(n/2) + cn \Rightarrow O(n)$ runtime

- Worst case: splitter is extremal element
  
  $T(n) \leq T(n − 1) + cn$

- General: splitter separates $cn$ elements, $(1 − \epsilon)n$ remain
  
  $T(n) \leq T((1 − \epsilon)n) + cn$

  $T(n) \leq cn[1 + (1 − \epsilon) + (1 − \epsilon)^2 + \ldots] \leq \frac{2n}{1 − \epsilon}$

**Randomized Splitters**

**Simple Idea:** Choose splitter uniformly at random.

Can’t hope to split into exactly half, but will see how long it takes to separate at least 1/4

- Call a splitter central if it separates 1/4 of elements
- $Pr[central\ splitter] = 1/2$ (must be in [1/4, 3/4] length)
- expected number of attempts $N$: $E[N] = 1/p = 2$

A central splitter reduces the set to 3/4 the size.

⇒ count phases of the algorithm, where $(\frac{1}{4})^{j+1}n \leq |S| \leq (\frac{3}{4})^jn$ (initially in phase 0)
Running Time Analysis

Let $X$ be a random variable counting the elementary steps done by the algorithm, and $X_j$ the steps in phase $j$.

\[ X = X_0 + X_1 + X_2 + \ldots \]

One iteration on size $n$: $cn$ steps.

Expected: two iterations per phase, size \( \leq (\frac{3}{4})^i n \)

\[ E[X] = \sum_j E[X_j] \leq \sum_j 2cn(\frac{3}{4})^j = 2cn \sum_j (\frac{3}{4})^j \leq 8cn \]

Application: Randomized Quicksort

Modified Quicksort:

- Choose random pivot, retry if not central (expected tries: 2)
- Divide array into $S^-$, $S^+$
- Recursively sort both
- Concatenate

Easy to analyze: expected work on each level is $\Theta(n)$

Can prove the same if continuing with any pivot

---

Weighted vertex cover

**Definition.** Given a graph $G = (V, E)$, a vertex cover is a set $S \subseteq V$ such that each edge in $E$ has at least one end in $S$.

**Weighted vertex cover.** Given a graph $G$ with vertex weights, find a vertex cover of minimum weight.

1. **Weight** $= 2 + 2 = 4$
2. **Weight** $= 11$

Pricing method

**Pricing method.** Each edge must be covered by some vertex.

Edge $e = (i, j)$ pays price $p_e \geq 0$ to use both vertex $i$ and $j$.

**Fairness.** Edges incident to vertex $i$ should pay $\leq w_i$ in total.

\[ \sum_{i \in i} p_e \leq w_i \]

**Fairness lemma.** For any vertex cover $S$ and any fair prices $p_e$: $\sum p_e \leq w(S)$.

**Pf.**

1. Each edge $e$ covered by at least one node in $S$.
2. Sum fairness inequalities for each node in $S$.

Pricing method example

- Price of edge $a \cdot b$
- Vertex weight
- $b$: tight

---

Easy to analyze: expected work on each level is $\Theta(n)$

Can prove the same if continuing with any pivot
**Pricing method: analysis**

**Theorem.** Pricing method is a 2-approximation for Weighted-Vertex-Cover.

**PF.**
- Algorithm terminates since at least one new node becomes tight after each iteration of while loop.
- Let $S = \{\text{set of all tight nodes upon termination of algorithm.}\}$
  - $S$ is a vertex cover: if some edge $(i,j)$ is uncovered, then neither $i$ nor $j$ is tight. But then while loop would not terminate.
- Let $S^*$ be optimal vertex cover. We show $w(S) \leq 2w(S^*)$.

\[
w(S) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{e \in E(i,j)} p_e \leq \sum_{i \in S} \sum_{e \in E(i,j)} p_e = 2 \sum_{i \in S} p_e \leq 2w(S^*).
\]

Knapsack Problem

- **n** items, weights $w_i$, values $v_i$, total capacity $W$
- **Goal:** maximize total value without exceeding capacity

<table>
<thead>
<tr>
<th>item</th>
<th>value</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

Knapsack Problem Approximation Algorithm

**Definition.** $OPT(i, v) = \min$ weight of a knapsack for which we can obtain a solution of value $\geq v$ using a subset of items $1, \ldots, i$

\[
OPT(i, v) = \begin{cases} 
OPT(j-1, w) & \text{if } u_j > w \\
\max \left\{ OPT(j-1, w), v_j + OPT(j-1, w - w_j) \right\} & \text{if } u_j \leq w
\end{cases}
\]

Good for small (maximum) weight.

Knapsack Problem Dynamic Programming v1

Choose from objects 1 through $j$, index by achieved weight

\[
OPT(j, w) = \begin{cases} 
OPT(j-1, w) & w_j > w \\
\max \left\{ OPT(j-1, w), v_j + OPT(j-1, w - w_j) \right\} & w_j \leq w
\end{cases}
\]

Knapsack Problem Dynamic Programming v2

**Definition.** $OPT(i, v) = \min \{OPT(i-1, v), w_i + OPT(i-1, v - v_i)\}$

\[
OPT(i, v) = \begin{cases} 
OPT(i-1, v) & \text{if } v \leq 0 \\
OPT(i-1, v - v_i) & \text{if } v > 0
\end{cases}
\]

- Running time is $O(nV)$ where $V$ is an upper bound on total value, e.g. $V = nV_{\text{max}}$
- Not polynomial in input size
- Polynomial if values are small integers

Knapsack Problem Approximation Algorithm

**Idea:** round all values (up) to coarser units and run dynamic programming algorithm

<table>
<thead>
<tr>
<th>item</th>
<th>value</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>934,221</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>5,956,342</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>17,810,013</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>21,217,800</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>27,734,384</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>item</th>
<th>value</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

The result will never exceed the weight capacity, but may lose some value due to rounding error.

Rounding Details

Round to nearest multiple of some integer $b$:
- $v_i$ = original value, 5,956,342
- $\bar{v}_i = \left\lfloor \frac{v_i}{b} \right\rfloor b$ = rounded value, 6,000,000
- $\left\lfloor \frac{v_i}{b} \right\rfloor = \text{value actually used in DP}$. 6
Analysis
▶ Let $S = \text{rounded DP solution}$, $S^* = \text{optimal solution}$
▶ We’ll show that $S$ gets nearly as much value as $S^*$:

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} \bar{v}_i \quad \text{round up}$$

$$\leq \sum_{i \in S} \bar{v}_i \quad S \text{ optimal for rounded values}$$

$$\leq \sum_{i \in S} (v_i + b) \quad \text{rounding amount} \leq b$$

$$\leq \sum_{i \in S} v_i + nb$$

$$\leq (1 + \epsilon) \sum_{i \in S} v_i$$

The last inequality is true if $nb \leq \epsilon \sum_{i \in S} v_i$. We need $b \leq \epsilon v_{\max}/n$.

Running Time
▶ Recall $b \leq \epsilon v_{\max}/n$.
We need $b \geq 1$, thus we can approximate if $\epsilon \geq n/v_{\max}$.
Require $\epsilon \geq \frac{c}{c-1} \frac{n}{v_{\max}}$ for some $c > 1$.

▶ $b = [\epsilon v_{\max}/n] > \frac{v_{\max}}{n} - 1 \geq \frac{v_{\max}}{n} (1 - \frac{1}{c}) = \frac{\epsilon v_{\max}}{cn}$, or $v_{\max}/b < cn/\epsilon$.

▶ $\bar{v}_{\max} = [v_{\max}/b] < v_{\max}/b + 1 < cn/\epsilon + 1$

▶ Running time for rounded DP is $O(n^2\bar{v}_{\max}) = O(n^3/\epsilon)$

Theorem: The rounding algorithm computes a solution whose value is within a $(1 + \epsilon)$ factor ($\epsilon > 0$) of the optimum in $O(n^3/\epsilon)$ time.