Structure

Cache Efficiency
  Introduction
  Cache Organization

Divide and Conquer
  Introduction
Cache Efficiency

Introduction

Background:
- Up to now we always counted the number of operations.
- Assuming this is a good measure for the runtime of an algorithm/tool.
- Today we will see examples where this is not suitable.
Example:

- We sum up all elements of an array $a$ of size $n$ in ... natural order:

  \[ \text{sum}(a) = a[1] + a[2] + \cdots + a[n] \]

- random order:

  \[ \text{sum}(a) = a[21] + a[5] + \cdots + a[8] \]
Python:

```python
def init(size):
    """Creates the dataset."""
    
    # use system time as seed
    random.seed(None)
    
    # set linear order as accessor
    order = [a for a in range(0, size)]
    
    # init array with random data
    data = [random.random() for a in order]
    
    return (order, data)
```
Python:

```python
def run(param):
    """Processes the dataset.""
    
    # unpack data
    (order, data) = param

    # init the sum value
    s = 0

    for index in order:
        s += data[index]

    return s
```
Figure: summing elements in linear order
def init(size):
    """Creates a randomly ordered dataset."""

    # use system time as seed
    random.seed(None)

    # set random order as accessor
    order = [a for a in range(0, size)]
    random.shuffle(order)

    # init array with random data
    data = [random.random() for a in order]

    return (order, data)
Figure: summing elements in random order
Conclusion:

- The number of operations is identical for both algorithms.
- Accessing elements in random order takes a lot longer (factor 10).
- The costs in terms of memory access are very different.
Cache Efficiency

CPU Cache

<table>
<thead>
<tr>
<th>Memory</th>
<th>0 0 0 0 0 0 0 0</th>
<th>0 0 0 0 0 0 0 0</th>
<th>0 0 0 0 0 0 0 0</th>
<th>0 0 0 0 0 0 0 0</th>
<th>0 0 0 0 0 0 0 0</th>
<th>0 0 0 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Register</td>
<td>0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Principle / organization:**

- Accessing one byte of the main memory takes $\approx 100 \text{ ns}$
- Accessing one byte of (L1-)cache takes $\approx 1 \text{ ns}$
- Accessing one or more byte/s of main memory loads a whole block $\approx 100 \text{ B}$ into the cache
- As long as this block is in the cache, it is not necessary to access the memory for bytes of this block
Cache Efficiency

CPU Cache

**Cache organization:**
- The (L1-)cache can hold multiple memory blocks
  - Cache lines \(\approx 100\text{kB}\)
- If the capacity is reached unused blocks are discarded
  - Least recently used (LRU)
  - Least frequently used (LFU)
  - First in first out (FIFO)
- Details of discarding not discussed today
**Cache Efficiency**

**Block Operations**

The system consists of slow and fast memory.

- The slow memory is divided into blocks of size $B$.
- The fast cache has size $M$ and can store $M/B$ blocks.

If data is not in fast memory, the corresponding block is loaded into the cache.

### Terminology:

- $B$ bytes
- $M$ bytes = $M/B$ blocks

---

 Majesty sizes:

- Memory
- Cache

---

December 2018  
Prof. Dr. Rolf Backofen – Bioinformatics - University Freiburg - Germany
### Terminology:
- The program defines which blocks are held in the cache.
- We use the number of block operations as runtime estimation.
- We ignore runtime costs of cache access / management.
**Cache Efficiency**

**Block Operations**

---

**good locality**

Memory:

```
0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0
```

**bad locality**

Memory:

```
0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0
```

**Figure:** comparison good / bad locality

---

**Accessing the cache $B$ times:**

- **Best case:** 1 block operation $\rightarrow$ good locality
- **Worst case:** $B$ block operations $\rightarrow$ bad locality
Additional factors:

- The following settings change only a small constant factor in number of block operations
  - Partitioning of the slow memory into blocks
  - Regardless of the block size: 1 bytes or 4 bytes or 8 bytes

Note:

- If the input size is smaller than $M$ we load the complete data chunk directly into the cache
- Cache handling is only interesting when the input size is greater than $M$
Typical values: (Intel© i7-4770 Haswell, WD© Blue 2TB)

- CPU L1 Cache: $B = 64\, \text{B}$, $M = 4 \times (32\, \text{kB} + 32\, \text{kB})$
- CPU L2 Cache: $B = 64\, \text{B}$, $M = 4 \times 256\, \text{kB}$
- CPU L3 Cache: $B = 64\, \text{B}$, $M = 8\, \text{MB}$
- Disk Cache: $B = 64\, \text{kB}$, $M = 64\, \text{MB}$

- Many operating systems use free system memory as disk cache
Cache Efficiency
Block Operations

Terminology:
- Block loads on CPU cache are called cache misses
- Block operations on disk cache are called IOs (input / output operations)
- These also fall under the term cache efficiency or IO efficiency
Example 1 - Linear order:

- We sum up all elements in natural order

\[
\text{sum}(a) = a[1] + a[2] + \cdots + a[n]
\]

- The number of block operations is \( \text{ceil} \left( \frac{n}{B} \right) \)

![Diagram showing cache and block read/write operations]

**Figure:** good locality of sum operation
Example 2 - Random order:

- We sum up all elements in random order

\[ \text{sum}(a) = a[21] + a[5] + \cdots + a[8] \]

- The number of block operations is \( n \) in the worst case

- This leads to a runtime factor difference of \( B \)

**Figure:** bad locality of sum operation
Generally the factor is substantially $< B$

- Even with a random order we access 4 neighboring bytes at once per `int` (int32_t)
- The next element might already be loaded into the cache
- If not $n \gg M$ this might occur with a high probability
QuickSort:

- **Strategy:** Divide and Conquer
- Divide the data into two parts where the “left” part contains all values ≤ the values in the right part
- Choose one element (e.g. the first one) as “pivot” element
- Ideally, both parts are the same size
- Both parts are sorted recursively

<table>
<thead>
<tr>
<th>p</th>
<th>list</th>
</tr>
</thead>
<tbody>
<tr>
<td>lower list</td>
<td>p</td>
</tr>
</tbody>
</table>

**Figure:** QuickSort with pivot element
Idea of Quicksort

- **At start:** pivot in first position, first re-arrange list such that left part contains smaller and right part larger elements
- Do required changes *in place*

- **End point:** $k$ is left to left-most element greater than pivot

  *swap position 0 (pivot) with $k$ (smaller than pivot)*
Python:

def quicksort(l, start, end):
    if (end - start) < 1:
        return

    i = start
    k = end
    piv = l[0]

    ...

def quicksort(l, start, end):
    ...

    while k > i:
        while l[i] <= piv and i <= end and k > i:
            i += 1
        while l[k] > piv and k >= start and k >= i:
            k -= 1

        if k > i: # swap elements
            (l[i], l[k]) = (l[k], l[i])
            (l[start], l[k]) = (l[k], l[start])
        quicksort(l, start, k - 1)
        quicksort(l, k + 1, end)
Number of operations for Quicksort:

- Let $T(n)$ be the runtime for the input size $n$

Assumptions:

- Arrays are always separated perfectly in the middle
- $n$ is a power-of-two and recursion depth is $k = \log_2 n$
Cache Efficiency
Block Operations - Quicksort

\[
T(n) \leq A \cdot n + 2 \cdot T \left( \frac{n}{2} \right)
\]

splitting in two parts

\[
\leq A \cdot n + 2 \left( A \cdot \frac{n}{2} + 2 \cdot T \left( \frac{n}{4} \right) \right)
\]

recursive sort

\[
= 2A \cdot n + 4 \cdot T \left( \frac{n}{4} \right)
\]

\[
\leq 3A \cdot n + 8 \cdot T \left( \frac{n}{8} \right)
\]

\[
\leq k \cdot A \cdot n + 2^k \cdot T(1)
\]

\[
= \log_2 n \cdot A \cdot n + n \cdot T(1)
\]

\[
\leq \log_2 n \cdot A \cdot n + n \cdot A \in O(n \log_2 n)
\]
Let $IO(n)$ be the number of block operations for input size $n$.

Assumptions as before but recursion depth is $k = \log_2 \frac{n}{B}$.
\[ \text{IO}(n) \leq A \cdot \frac{n}{B} + 2 \cdot \text{IO}(\frac{n}{2}) \]

splitting in two parts

\[ \leq A \cdot \frac{n}{B} + 2 \left( A \cdot \frac{n}{2B} + 2 \cdot \text{IO}(\frac{n}{4}) \right) \]

recursive sort

\[ \leq 2 \cdot A \cdot \frac{n}{B} + 4 \cdot \text{IO}(\frac{n}{4}) \]

\[ \leq 3 \cdot A \cdot \frac{n}{B} + 8 \cdot \text{IO}(\frac{n}{8}) \]

\[ \leq \ldots \]

\[ \leq k \cdot A \cdot \frac{n}{B} + 2^k \cdot \text{IO}(\frac{n}{2^k}) \]

\[ = \log_2(\frac{n}{B}) \cdot A \cdot (\frac{n}{B}) + n/B \cdot \text{IO}(B) \]

\[ \leq \log_2(\frac{n}{B}) \cdot A \cdot (\frac{n}{B}) + A \cdot \frac{n}{B} \quad \in \quad O \left( \frac{n}{B} \cdot \log_2 \left( \frac{n}{B} \right) \right) \]
Concept:

- Divide the problem into smaller subproblems
- Conquer the subproblems through recursive solving. If subproblems are small enough solve them directly
- Connect all solutions of the subproblems to the solution of the full problem
- Recursive application of the algorithm to ever smaller subproblems
- Direct solving of sufficiently small subproblems
Divide and Conquer
Introduction - Python

- Function `solve` for solving a problem of size \( n \)

```python
def solve(problem):
    if n < threshold:
        return solution # solve directly
    else:
        # divide problem into subproblems
        # P1, P2, ..., Pk with k >= 2
        S1 = solve(P1)
        S2 = solve(P2)
        ...
        Sk = solve(Pk)

        # combine solutions
        return S1 + S2 + ... + Sk
```
Divide and Conquer:  
- Can help with conceptual hard problems  
- Solution of the trivial problems has to be known  
- Dividing into subproblems has to be possible  
- Combination of solutions has to be possible
Features:

- Realization of efficient solutions
  - If trivial solution is \( \in O(1) \)
  - And separation / combination of subproblems is \( \in O(n) \)
  - And the number of subproblems is limited
  - The runtime is \( \in O(n \cdot \log n) \)

- Suitable for parallel processing
  - Parallel processing of subproblems possible since subproblems are independent of each other
Definition of the trivial case:

- Smaller subproblems are elegant and simple
- On the other hand the efficiency will be improved if relatively big subproblems can be solved directly
- Recursion depth should not get too big (stack / memory overhead)
Division in subproblems:
- Choosing the number of subproblems and the concrete allocation can be demanding

Combination of solutions:
- Typically conceptionally demanding
Example - Maximum Subtotal Input:

- Sequence $X$ of $n$ integers

Output:

- Maximum sum of related subsequence and its index boundary

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>31</td>
<td>-41</td>
<td>59</td>
<td>26</td>
<td>-53</td>
<td>58</td>
<td>97</td>
<td>-93</td>
<td>-23</td>
<td>84</td>
</tr>
</tbody>
</table>

Output: sum: 187, start: 2, end: 6
Application:
- Maximum profit of buying and selling shares

Figure: stock value over time
Naive solution (brute force)

```python
def maxSubArray(X):
    # Store sum, start, end
    result = (X[0], 0, 0)
    for i in range(0, len(X)):
        for j in range(i, len(X)):
            subSum = 0
            for k in range(i, j + 1):
                subSum += X[k]
            if result[0] < subSum:
                result = (subSum, i, j)
    return result
```
Divide and Conquer
Example - Maximum Subtotal - Python

Runtime - Upper bound

```python
def maxSubArray(X):
    result = (X[0], 0, 0)
    # n loops -> O(n)
    for i in range(0, len(X)):
        # max n loops -> O(n)
        for j in range(i, len(X)):
            # max n loops -> O(n)
            subSum = sum(X[i:j+1])
            if result[0] < subSum: # O(1)
                result = (subSum, i, j)
    return result
```
Upper bound:

- Three nested loops
- Each loop with runtime $O(n)$
- Algorithm runtime of $O(n^3)$
Divide and Conquer
Example - Maximum Subtotal - Runtime

Lower bound:

Table: Operations

<table>
<thead>
<tr>
<th>i</th>
<th>Additions</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{n}{3} \in O(n)$</td>
<td>$\frac{n}{3} \in O(n)$</td>
<td>$\frac{n}{3} \in O(n)$</td>
</tr>
</tbody>
</table>

- We iterate at least $\frac{n}{3}$ values for $i$
- For each $i$ we iterate at least $\frac{n}{3}$ values for $j$
- For each $j$ we have at least $\frac{n}{3}$ additions
- We need at least $T(n) = \left(\frac{n}{3}\right)^3 \in \Omega(n^3)$ steps
Runtime:

- With $T(n) \in O(n^3)$ and $T(n) \in \Omega(n^3)$ we know:

$$T(n) \in \Theta(n^3)$$

- It is hard to solve the problem in a worse way ...
Current approach:

- Calculating the sum for range from \( i \) to \( j \) with loop

\[
S_{i,j} = X[i] + X[i + 1] + \cdots + X[j]
\]

Better approach:

- Incremental sum instead of loop

\[
S_{i,j+1} = S_{i,j} + X[j+1] \in O(1) \quad \text{instead of} \quad \in O(n)
\]
Better solution:

```python
def maxSubArray(X):
    result = (X[0], 0, 0)
    # n loops -> O(n)
    for i in range(0, len(X)):
        subSum = 0
        # max n loops -> O(n)
        for j in range(i, len(X)):
            subSum += X[j] # O(1)
            if result[0] < subSum: # O(1)
                result = (subSum, i, j)
    return result
```

- Runtime $\in O(n^2)$
Divide and Conquer:

- Split the sequence in the middle
- Solve left half of the problem
- Solve right half and combine both solutions into one
- Maximum might be located in left half \((A)\) or right half \((B)\)
- Problem: Maximum can overlap the split
- To solve this case we have to calculate \(r_{\text{max}}\) and \(l_{\text{max}}\)
- The overall solution is the maximum of \(A, B\) and \(C\)
Divide and Conquer
Example - Maximum Subtotal

Principle - Divide and Conquer:

- Small problems are solved directly: $n = 1 \Rightarrow \text{max} = X[0]$
- Bigger problems are partitioned into two subproblems and solved recursively. Subsolutions A and B are returned
- To determine subsolution C, rmax and lmax for the subproblems are computed
- The overall solution is the maximum of A, B and C
def maxSubArray(X, i, j):
    if i == j:  # trivial case
        return (X[i], i, i)

# recursive subsolutions for A, B
m = (i + j) // 2
A = maxSubArray(X, i, m)
B = maxSubArray(X, m + 1, j)

# rmax and lmax for  corner case C
C1, C2 = rmax(X, i, m), lmax(X, m + 1, j)
C = (C1[0] + C2[0], C1[1], C2[1])

# compute solution from results A, B, C
return max([A, B, C], key=lambda i: i[0])
Further Literature

General


Further Literature

Caching

[Wik] Cache